RL Project Report

Peng Tang
517020910038
tttppp@sjtu.edu.cn

Abstract

In this project, we use two kinds of model-free RL methods, namely value-based RL and policy-based RL, to play some games of gym. For valued-based RL algorithm, we try several different versions of DQN, including Double-DQN, Dueling-DQN and Prioritized-DQN. Finally, we find that the original DQN is best one and We can not find the reason. For policy-based RL algorithm, we use TD3 to implement it. Meanwhile, we utilize some methods to improve the performance of our models, such as vision transformation.

1 Introduction

This project is to use gym as experiment environment, especially, four games of Atari and four games of MuJuCo. What's more, the action space of all fours games of Atari is discrete, while MuJuCo is continuous.

Value-based methods strive to fit action value function or state value function, e.g. MC, SARSA and Q-learning. DQN is very suitable for discrete action space, so we choose it to play the Atari games. What's more, there are many advanced DQN methods, such as Double-DQN, Dueling-DQN, Prioritized-DQN, NoisyNet-DQN and so on.

Policy-based methods strive to output the action directly according current state without computing the value of state or action, e.g. A2C, A3C and DDPG. DDPG is an off-policy method(more efficient) and achieves remarkable performance in tasks with continuous action space. Therefore, we choose TD3, an advanced version of DDPG, to play the MuJuCo games.

2 Algorithm Design

2.1 Value-based RL Algorithm

For valued-based RL algorithm, we try following different versions of DQN.

2.1.1 DQN

DQN generate Q values through neural network, which overcomes the disadvantage of ordinary Q-Learning algorithm, which need a large space to store all Q values of corresponding states and actions. Especially, we implement DQN by constructing two neural networks, target network and evolution network. The two networks are almost same except parameters. Whats more, we need a replay buffer to store some transitions for training.

When training the evolution network, we firstly get a batch of transitions from the replay buffer randomly. Then we input the states of the batch to evolution network and get corresponding $Q(s, a)$ of the batch. To get the $Q^{Target}(s, a)$, we input the next-states of the batch to target network and get the $\max_{a'} Q(s', a')$. We can compute the $Q^{Target}(s, a)$ according to Q-learning algorithm as

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

equation 1.

$$Q^{Target}(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{1}$$

Then we can compute the loss function for evolution network according to equation 2.

$$Loss(w) = E[(Q^{Target}(s,a,w) - Q(s,a,w))^2] \tag{2}$$

With this loss function, we can train the evolution network. And we update the target network with the parameters of evolution network after some episodes. To illustrate the main idea of DQN in our model more clearly, we use Figure 1 to show it.

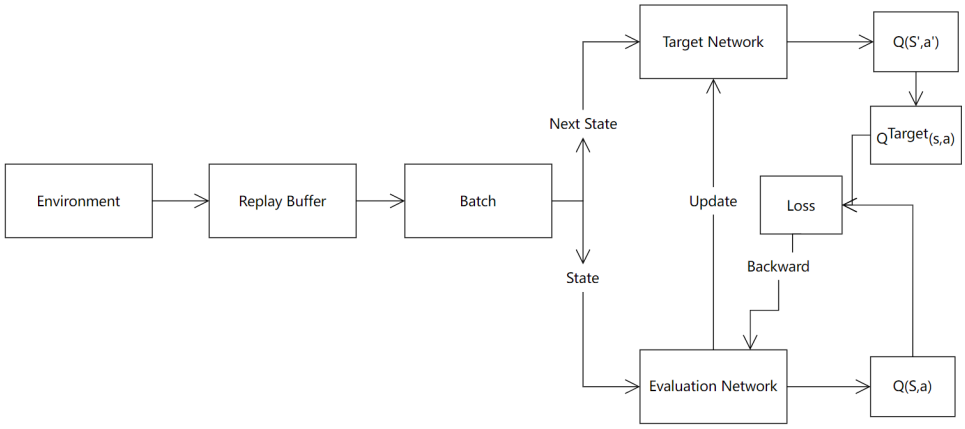

Figure 1: DQN training process

The DQN algorithm is shown as Algorithm 1(Mnih et al. [2013]).

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode=1, M **do**
  Initialize sequenece $s_1 = x_1$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}) =$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for nonterminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

2.1.2 Double DQN

The DQN algorithm easily over-estimate action values and these over-estimations usually have a bad impact on performance. According to the paper *Deep Reinforcement Learning with Double Q-learning*(van Hasselt et al. [2015]), we use Double-DQN algorithm to reduce these over-estimations and improve the performance of our agent.

The main difference between DQN and Double-DQN is the method of computing $Q^{Target}(s,a)$. The equation 3 shows the $Q^{Target}(s,a)$ computing of DQN and the equation 4 shows the

$Q^{Target}(s,a)$ computing of Double-DQN.

$$Q^{Target}(s,a,w) = r + \gamma \max_{a'} Q(s',a',w) \tag{3}$$

$$Q^{Target}(s,a,w) = r + \gamma Q(s', \arg\max_{a'} Q(s',a',w'),w) \tag{4}$$

Where w represents the parameters of target network and w' represents the parameters of evolution network. From these equations, we can find that Double-DQN using the evolution network to get the action for s' rather than target network. In this way, Double-DQN use two networks' information to compute Q values.

2.1.3 Priority Replay Buffer

Replay buffer is used to store experiences from the past. In our previous model, we chose a batch of transitions from replay buffer randomly. However, the importance of different transitions should be different. These transitions with higher prediction loss should be more important and be more likely to be selected. In this way, the model can reach our goal with faster speed. In practice, we give every transition a corresponding priority. The higher the priority of a transition is, the more possible it will be sampled.

According to the paper *Prioritized Experience Replay*(Schaul et al. [2016]), We can define the priority of transitions by using the TD-error, which is the difference between the $Q^{Target}(s,a)$ and $Q(s,a)$. Thus, we define the priority of transition i as equation 5

$$p_i = \begin{cases} \max_{t<i} p_t, & case \quad 1 \\ (|\delta_i| + \epsilon)^\alpha, & case \quad 2 \end{cases} \tag{5}$$

In case 1, transition i is added to the replay buffer for the first time. In this case, because transition i is the newest, it should be important. Thus, we set its priority to maximum so that it can be selected with high probability.

In case 2, $\delta_i = Q^{Target}(s,a) - Q(s,a)$. ϵ is a small positive constant number that prevents the priority of transition i being zero. And $\alpha (0 \leq \alpha \leq 1)$ determines the importance of error. $\alpha = 0$ is the uniform case.

With the priority, we define the probability of sampling transition i as equation 6

$$P(i) = \frac{p_i}{\sum_k p_k} \tag{6}$$

What's more, with probability, we define importance-sampling(IS) weights(equation 7) to correct the bias caused by prioritized replay buffer.

$$w_j = \frac{(N * P(j))^{-\beta}}{\max_j w_j} \tag{7}$$

Where β linearly increase from a initial number β_0 to 1. With IS weights, we redefine loss function as equation 8

$$Loss = E[W * (Q^{Target} - Q)^2] \tag{8}$$

To improve the sampling efficiency, we use a binary tree data structure, sum tree, to store transitions. The value of the leaves in the sum tree is the priority of transitions and the parent's value is the sum of its children. Thus, the value of the root is the total sum of all transitions' priority. Using sum tree, we can sample, insert and update buffer with $O(\log n)$. We can construct sum tree by using an array. In this way, $LeftChild_{index} = 2 * Parent_{index} + 1$ and $RightChild_{index} = LeftChild_{index} + 1$. Therefore, we use a replay buffer D to store all transitions and a sum tree T to store corresponding priority. If the size of D is N, then the size of T is $2N - 1$ because we need N leaves in T. Thus, the corresponding priority of transition i in D is the value of $T[i + N - 1]$.

With above analysis, we can select a transition with Algorithm 2.

Algorithm 2 Get a transition from replay buffer D according to sum tree T

Input: priority number x
Output: transition t, its priority p and its index t_{index} in T

Set t_{index} to be 0
while $2 \times t_{index} + 1$ is smaller than the length of T **do**
  Left child index $l \Leftarrow 2 \times t_{index} + 1$
  Right child index $r \Leftarrow l + 1$
  if $x \leq T[l]$ **then**
    $t_{index} \Leftarrow l$
  else
    $x \Leftarrow x - T[l]$
    $t_{index} \Leftarrow r$
  end if
end while
Compute the index of transition t in D by setting $d_{index} \Leftarrow t_{index} - N + 1$, N is the size of D
$t \Leftarrow D[d_{index}]$
$p \Leftarrow T[t_{index}]$

Using Algorithm 2, we can sample a minibath with Algorithm 3.

Algorithm 3 Get a minibatch from replay buffer D according to sum tree T

Input: batch size k
Output: Batch B, its index B_{index} and its weight W

Initialize empty array B, B_{index}, and W with capacity k
Divide $[0, T[0]]$ into k ranges equally
for $i = 1$ to k **do**
  Randomly choose a number x_i from range i
  Call Algorithm 2 to get transition t_i, its priority p_i and its index $t_{i_{index}}$ by inputting x_i
  Compute probability P_i according to equation 6 with p_i
  Compute weight w_i according to equation 7 with P_i
  Store t_i in B, w_i in W and $t_{i_{index}}$ in B_{index}
end for

In order to update the sum tree when we change a leaf's priority, we define Algorithm 4.

Algorithm 4 Update sum tree T

Input: transition's index t_{index} in T and its new priority p

$c \Leftarrow p - T[t_{index}]$
$T[t_{index}] \Leftarrow p$
while $i \neq 0$ **do**
  $i \Leftarrow \frac{t_{index}-1}{2}$
  $T[t_{index}] \Leftarrow T[t_{index}] + c$
end while

With Algorithm 4, we define Algorithm 5 to store a new transition to the replay buffer D and its priority to the sum tree T.

Algorithm 5 Store a transition to replay buffer D and its priority to sum tree T.

Input: transition t

Set the priority p of transition t to be the max priority in T.
If p is zero, set it to be 1
Compute its index t_{index} in T according to the current pointer d_{index} of D. $t_{index} \Leftarrow d_{index} + N - 1$, N is the size of D.
$D[d_{index}] \Leftarrow t$
Call Algorithm 4 to update T by inputting t_{index} and p
$d_{index} \Leftarrow (d_{index} + 1) \mod N$

2.1.4 Dueling DQN

In Double-DQN and Prioritized-replay-DQN, we optimize the calculation of target Q value and sampling process respectively. In Dueling-DQN, we aim at optimizing the structure of neural network. Dueling-DQN(Wang et al. [2016]) divides the original Q network into two parts. First part, denoted as $V(S, w)$, is only related with current state. Second part, denoted as $A(S, A, w)$, is related with both current state and actions. And We can represent our Q value as follows:

$$Q(S, A) = V(S, w) + A(S, A, w)$$

This method is more efficient in using the sampled data. For example, when we sample one pair (S, A), we can also adjust $Q(S, A')$ by change $V(S, w)$. In addition, to prevent our model from degenerating into original model, we add some constrains on $A(S, A, w)$ to limit $\sum_a A(S, a, w)$ to 0. We get the final expression:

$$Q(S, A) = V(S, w) + (A(S, A, w) - \frac{1}{C} \sum_{a' \in \mathcal{A}} A(S, a', w))$$

where $\mathcal{A}$ is the action space and C is the size of action space $\mathcal{A}$.

2.2 Policy-based RL Algorithm

We choose the TD3, the advanced version of DDPG, as our policy-based RL Algorithm.

2.2.1 DDPG

DDPG(Lillicrap et al. [2015]) is a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces. DDPG can be considered as a continuous action version of DQN.

$$DQN: \quad a^* = \arg\max_a Q(s, a)$$
$$DDPG: \quad a^* = \mu_\theta(s), \quad Q(s, a^*) = \max_a Q(s, a)$$

Where $\mu_\theta(s)$ is a deterministic policy which can directly give the action that maximizes $Q(s, a)$.

Similar to DQN, DDPG uses two actor-critic networks to train the model. Figure 2 shows the framework of DDPG.

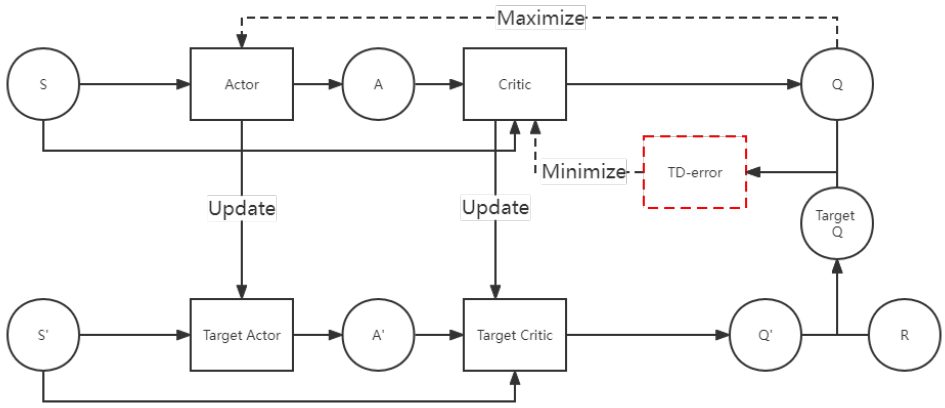

Figure 2: DDPG framework

2.2.2 TD3

To gain better performance, TD3(Fujimoto et al. [2018]) uses some tricks to improve DDPG, mainly including following tricks.

1. **Clipped Double Q-Learning:** Like DQN, DDPG has overestimation problem. To reduce it, TD3 use two critics to compute the target Q-values(Equation 9).

$$y = r + \gamma \min_{i=1,2} Q_i(s', \pi(s')) \tag{9}$$

   Where Q_1 and Q_2 is the two critics; π is the actor.

2. **"Delayed" Policy Updates:** TD3 updates the policy (and target networks) less frequently than the Q-function, which can make the Q-function converge more easily.

3. **Target Policy Smoothing:** TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action(Equation 10).

$$y = r + \gamma Q(s', \pi(s') + \epsilon), \epsilon \sim clip(N(0, \sigma), -c, c) \tag{10}$$

2.2.3 Final Algorithm

With DDPG and TD3, we can get our final policy-based model as Algorithm6

Algorithm 6 Policy-Based Algorithm

Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters $\theta_1, \theta_2, \pi_\phi$.
Initialize target networks $T_{\theta_1'}, T_{\theta_2'}, \pi_{\phi'}^T$ with $\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize replay buffer D with size N, batch size K,the variance of action noise σ, action noise arrange c, max value of action a_m, discount factor γ, updating frequent F, and slow-moving update rate τ
$t \leftarrow 1$
for $episode = 1$ to $Max\_Episode$ **do**
  Get initial state s from game
  while $True$ **do**
    Select action $a \sim \pi_\phi(s) + \epsilon, \quad \epsilon \sim N(0, \sigma)$
    $a \leftarrow clip(a, -a_m, a_m)$
    Get next step information from the environment through action a ,$(s', r, done) = env(a)$, s' is next state and r is reward
    Store transition $(s, a, r, s', done)$ in D
    if D is filled **then**
      Randomly sample mini-batch of K transitions (s, a, r, s', d) from D
      $a' \leftarrow \pi_{\phi'}(s') + \epsilon, \epsilon \sim clip(N(0, \sigma), -c, c)$
      $a' \leftarrow clip(a', -a_m, a_m)$
      $y \leftarrow r + (1 - d)\gamma \min_{i=1,2} T_{\theta_i'}(s', a')$
      Update critics $\theta_i \leftarrow \arg\min_{\theta_i} K^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
      if $t \mod F$ **then**
        Update ϕ by maximizing the $Q_{\theta_1}(s, \pi_\phi(s))$ with policy gradient
        Update target Networks:
        $\theta_i \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
        $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
      end if
    end if
    $s \leftarrow s'$
    $t \leftarrow t + 1$
    if $done$ **then**
      break
    end if
  end while
end for

3 Experiments And Results

3.1 Value-based Model

We implement above different DQNs and test them on four given environments of Atari. Some useful information about the four environments is shown as Table 1. Because the observation of

Table 1: Information of Atari Games

	Observation Shape	Observation Range	Action Shape	Action Range	Reward Range
VideoPinball-ramNoFrameskip-v4	$(128,)$	$(-\infty, \infty)$	1	$\{0, 1, \cdots, 8\}$	$(-\infty, \infty)$
BreakoutNoFrameskip-v4	$(260, 130, 3)$	$(-\infty, \infty)$	1	$\{0, 1, \cdots, 3\}$	$(-\infty, \infty)$
PongNoFrameskip-v4	$(260, 130, 3)$	$(-\infty, \infty)$	1	$\{0, 1, \cdots, 6\}$	$(-\infty, \infty)$
BoxingNoFrameskip-v4	$(260, 130, 3)$	$(-\infty, \infty)$	1	$\{0, 1, \cdots, 17\}$	$(-\infty, \infty)$

VideoPinball-ramNoFrameskip-v4 is vector, we use four liner layers and three activate function layers($Relu()$) to construct the network. For **BreakoutNoFrameskip-v4**, **PongNoFrameskip-v4** and **BoxingNoFrameskip-v4**, because their observations are 210×160 RGB images, we use the following steps(Mnih et al. [2013]) to preprocess them.

- Convert the RGB image into Greyscale.
- Resize the Greyscale to 84×84.

After preprocessing, we stack four consecutive frames to construct the state $s_t = (x_{t-3}, x_{t-2}, x_{t-1}, x_t)$. Finally the shape of the state we get is $(4, 84, 84)$. Because the inputting state is image, we construct CNN(three Conv2D layers and two linear layers) as the network.

In our experiments, we try Dueling DQN, Double DQN, priority buffer DQN and original DQN, we find the original DQN get the best performance. By training the original DQN on three environments, we gain the training process as Figure 3.

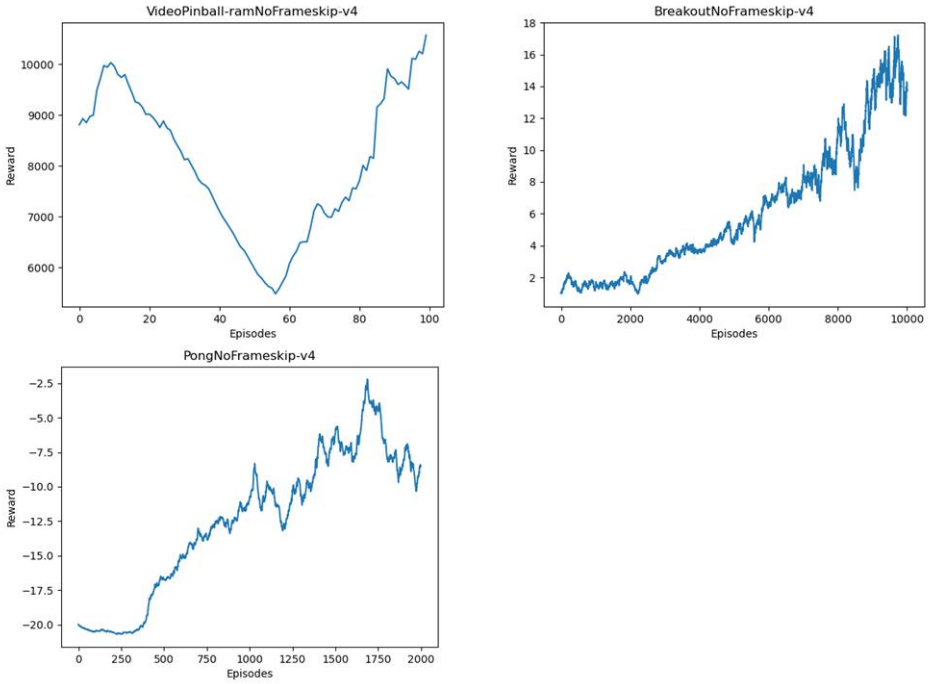

Figure 3: Valued-based Model Training Process

Then, by testing the trained model on each environment for 50 episodes, we gain the results shown as Figure 4 and Table 2.

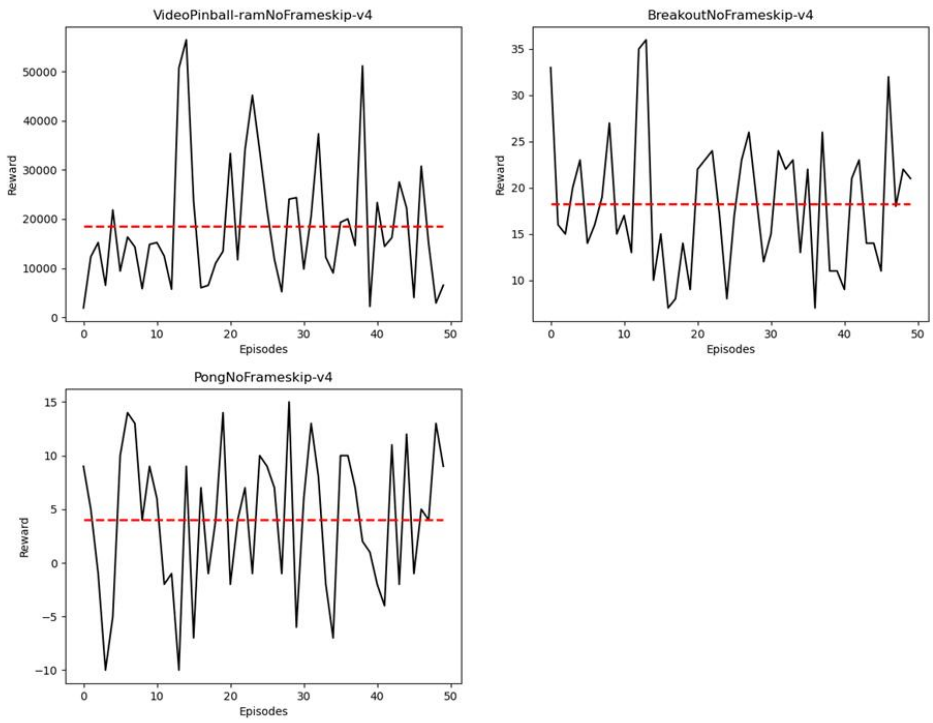

Figure 4: Testing Results of Valued-based Model

Table 2: Average Rewards of Valued-based Model

Environment	VideoPinball-ramNoFrameskip-v4	BreakoutNoFrameskip-v4	PongNoFrameskip-v4
Average Reward	18488.76	18.24	4.2

3.2 Policy-based Model

We implement Algorithm 6 and test it on four given environments of MuJuCo. Some useful information about the four environments is shown as Table 3.

Table 3: Information of MuJuCo Games

	Observation Shape	Observation Range	Action Shape	Action Range	Reward Range
Hopper-v2	$(11,)$	$(-\infty, \infty)$	$(3,)$	$(-1, 1)$	$(-\infty, \infty)$
Humanoid-v2	$(376,)$	$(-\infty, \infty)$	$(17,)$	$(-0.4, 0.4)$	$(-\infty, \infty)$
Ant-v2	$(111,)$	$(-\infty, \infty)$	$(8,)$	$(-1, 1)$	$(-\infty, \infty)$
HalfCheetah-v2	$(17,)$	$(-\infty, \infty)$	$(6,)$	$(-1, 1)$	$(-\infty, \infty)$

In our work, we choose DNN to construct Actor Network and Critic Network. Because both observation and action of all four environments are vector, we use three linear layers and three activate function layers(two $Relu()$ and one $Tanh()$) to build Actor and three linear layers and two activate function layers(two $Relu()$) to build Critic.

By training our model on four environments, we gain the training process as Figure 5.

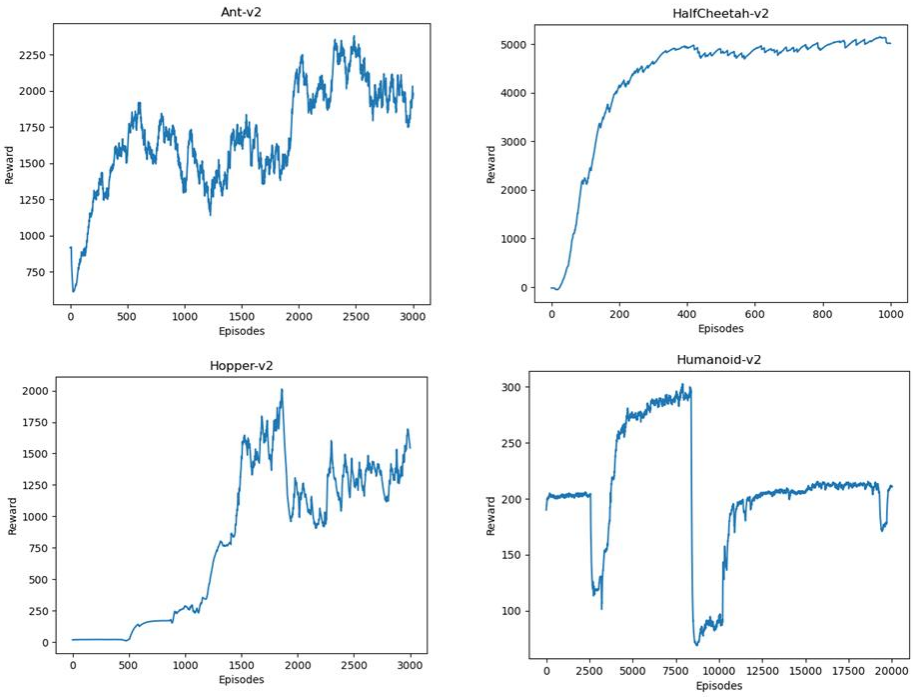

Figure 5: Policy-based Model Training Process

Then, by testing the trained model on each environment for 50 episodes, we gain the results shown as Figure 6 and Table 4.

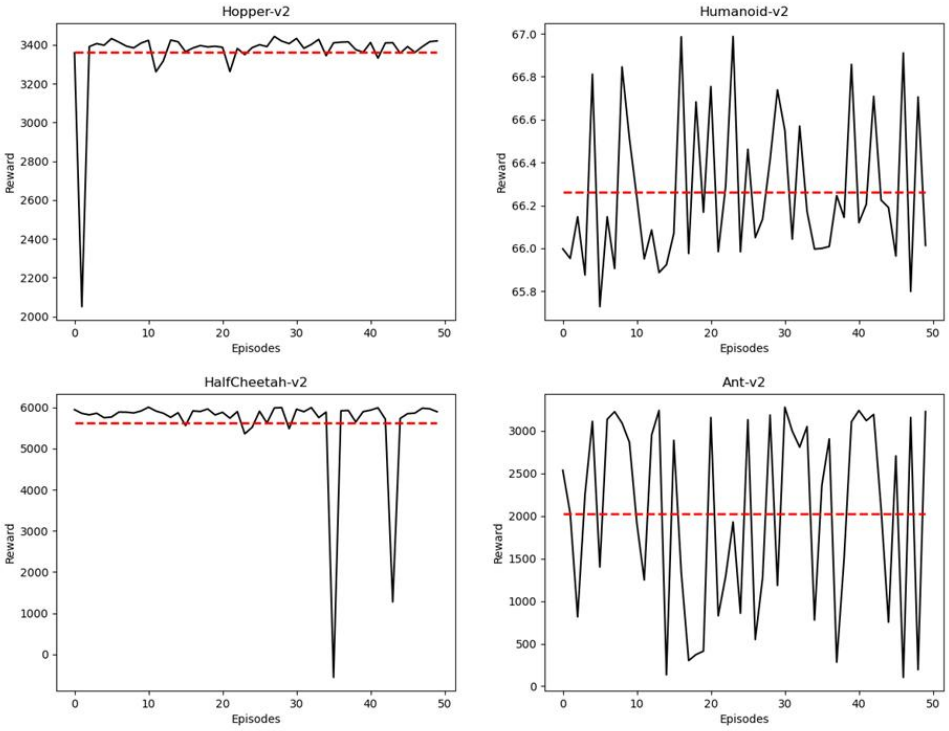

Figure 6: Testing Results of Policy-based Model

Table 4: Average Rewards of Policy-based Model

Environment	Hopper-v2	HalfCheetah-v2	Humanoid-v2	Ant-v2
Average Reward	3362.48	5619.16	66.26	2028.19

4 Discussion and Analysis

4.1 Valued-based Algorithm

- From Figure 3, 4 and Table 2, we can see that only VideoPinball-ramNoFrameskip-v4 can reach good performance. We think this is because its observation is the simplest, the simple network can also recognize it.
- Except VideoPinball-ramNoFrameskip-v4, the other two games performs badly. The increasing speed of them is too slow. Ten million steps training just gets a little improvement. This may be because the network or the hyper-parameter is not suitable. Although we try some different hyper-parameters and networks, we don't find a good one. And we also try the **ResNet18**(a remarkable network for image recognition)(He et al. [2015]), but the training speed is too slow to wait. Because the state consists of four consecutive frames, maybe the RNN is a suitable network.
- The three advanced DQNs get very bad performance. Like Figure 7, the reward keeps decreasing along the training process and we can't find the reason.

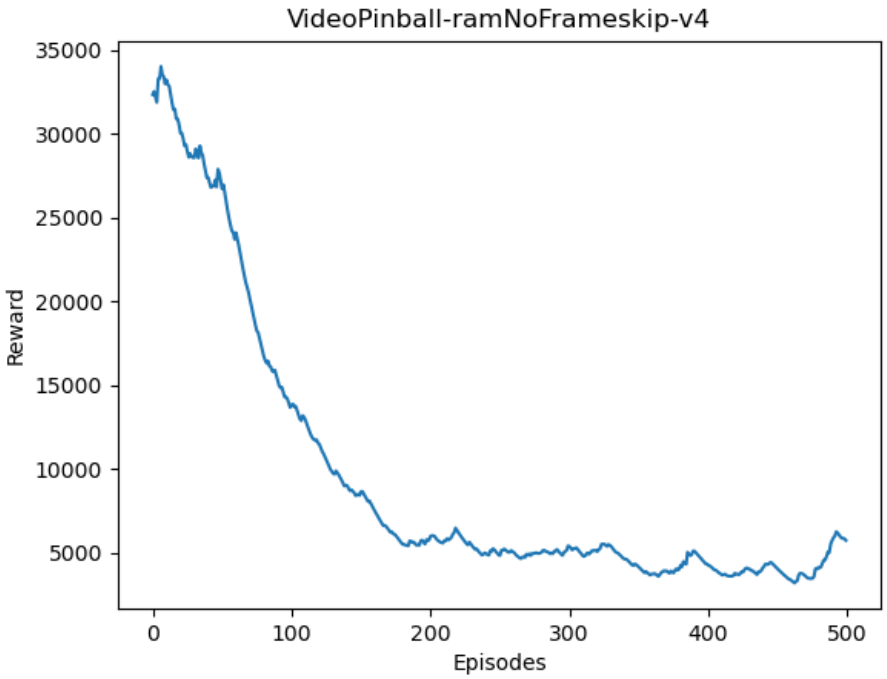

Figure 7: Bad Result

4.2 Policy-based Algorithm

- From Figure 5, 6 and Table 4, we can see that the performance of the model on HalfCheetah-v2 is the best, while on Humanoid-v2 is the worst. Maybe this is because we use same network architecture on all environments, but different environments should have different networks. What's more, Ant-v2 and Hopper-v2 can't converge in our experiment.

- In the process of experiment, we find that hyper-parameters have a great influence on the performance of the model. When training the model on Ant-v2 for the first time, the reward even decreases as training process. Then, by tuning some hyper-parameters, we gain much better performance. And because there are many hyper-parameters in this algorithm, it is difficult to find suitable hyper-parameters for each environment.
- Compared to the results in the original paper(Fujimoto et al. [2018]), our results isn't good enough. Maybe we can try different networks and parameters to gain good performance.

5 Conclusion

In this project, we implement a value-based algorithm(advanced DQN) and a polucy-based algorithm(TD3) and test them on given environments. Although we don't reach good performance on all environments as we consider, we learn some experience in this process.

- It is difficult to find a network which is suitable for several environments.
- Hyper-parameters and initial states have a big influence on the performance of model.
- The training time is very long for a little complex environment.

All in all, Reinforce Learning is a difficult area.

References

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. *arXiv e-prints*, art. arXiv:1802.09477, February 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL `http://arxiv.org/abs/1512.03385`.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, art. arXiv:1509.02971, September 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.