Towards The Glory of Deterministic Control: DQN, DDPG, and Their Variants

Zhengxiang Huang

520021910014

Abstract

Deep Reinforcement Learning is the most intriguing and promising topic in the discipline of control theory. Previous works have already shown the great potential of Deep RL on various tasks to achieve human-level control. In this project, we focus on 2 state-of-the-art deterministic control algorithm, DQN and DDPG, and also their variants. These 2 models are first implemented and tested on 4 Atari 2600 games and 4 MUJOCO environments. To fully exploit the power of the models, some tricks are borrowed and introduced from several recent works to improve the model performance. Finally, a brief analysis on the inherent advantage of deterministic control over stochastic policy control is also derived via the reduction argument at the end. My code and demo videos are available on Github.

https://github.com/huangzhengxiang/RL_Project.git

Keywords: **Deep Reinforcement Learning, DQN, DDPG, Deterministic Control**

1 Introduction

1.1 Background

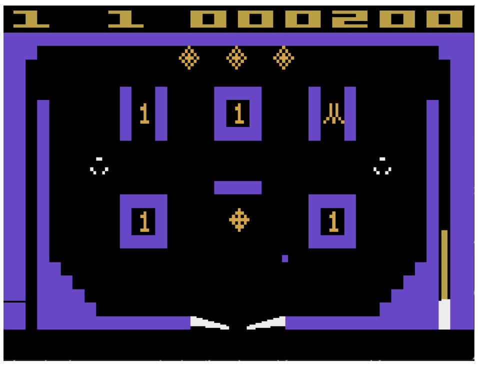
Generally speaking, machine learning can be categorized in to 3 main categories: supervised learning, unsupervised learning, and reinforcement learning. Unlike supervised learning, which heavily depends on labeled data, and unsupervised learning, which relies on the prior model hypotheses, either in need of human labor or model prior, reinforcement learning, on the other hand, searches the action space on its own, **learn by trail and error from delayed rewards** [7], the closest to natural intelligence among all artificial intelligence. It's probability the best way to create a real intelligent agent who can keeping learning forever through interacting with the environment.
Reinforcement Learning has quite a long history compared to deep learning. Traditional tabulation methods such as Sarsa, Q-Learning [7], and even such advanced tricks as Doube Q-Learning [8] and Deterministic Policy Gradient (DPG) [6] have long been devised and achieved good results in many applications. However, when the dimension of observation space and the action space surges, the cost of traditional tabulation simply explodes. New methods are in demand. Fortunately, recent development in deep learning helps. Deep Reinforcement Learning becomes the solution to high dimensionality. Such methods as DQN and DDPG reinforce the power of the original Q-Leaning and DPG (Deterministic Policy Gradient) and improve their performance by large. In these recent works we see the bright future in Deep RL. To sum up, it's high time to investigate Deep RL.
In this project, we only focus on 2 state-of-the-art model-free methods: **DQN** [5] and **DDPG** [3]. For DQN, it's tuned and tested on **Atari 2600** games, while for DDPG, it's implemented and tested in **MUJOCO** (Multi-Joint dynamics with Contact) environment. Atari games are representative game environments for continuous and high-dimension observation space and discrete and small

action space. MUJOCO is a famous environment where observation and action space are both continuous and low-dimensional.
To deal with the high dimensionality of observation space, DQN replaces traditional linear approximator with **convolutional neural networks (CNN)**. The action is selected simply via finding the maximum of $\max_a Q(s, a)$, thanks to the discretion of action space. To further deal with the continuity of the observation, DDPG introduces **policy networks** to dynamically and directly estimate the maxima of $Q(s, a)$ instead of quantizing the action space.
Besides introducing, implementing, and testing these 2 methods, we also discuss **their variants**, such as Double DQN [9], Dueling DQN [10], [2], and TD3 (Twin Delayed DDPG) [1]. We will see the power and glory of **deterministic control**, and how they may excel and tower over their stochastic counterparts such as **A3C**. The comparison between DDPG and A3C [4] is also provided.

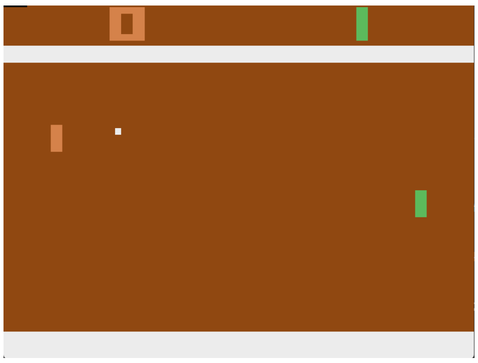
1.2 Problem Formulation

We implement DQN, DDPG, and A3C with Python3 and Pytorch, and train and test them in *gymnasium or gym* environment. The experiments consist of the following.
Testing DQN, Double-DQN, and Dueling DQN in the following assigned Atari 2600 games. We didn't formally implement rainbow, but we borrow some ideas from it.

1. VideoPinball-ramNoFrameskip-v4
2. BreakoutNoFrameskip-v4
3. PongNoFrameskip-v4
4. BoxingNoFrameskip-v4


(a) VideoPingball

(b) Breakout


(c) Pong

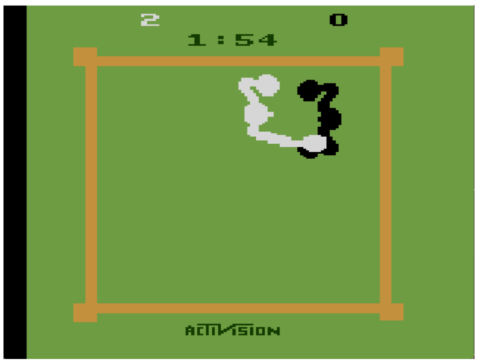

(d) Boxing

Figure 1: Screen Shots of 4 Atari Games

We test DDPG, Twin Delayed DDPG (TD3), and A3C in the following 4 MUJOCO tasks. Also, we didn't formally implement TD3, but we borrow some tricks from it.

1. Hopper-v2
2. Humanoid-v2
3. HalfCheetah-v2
4. Ant-v2

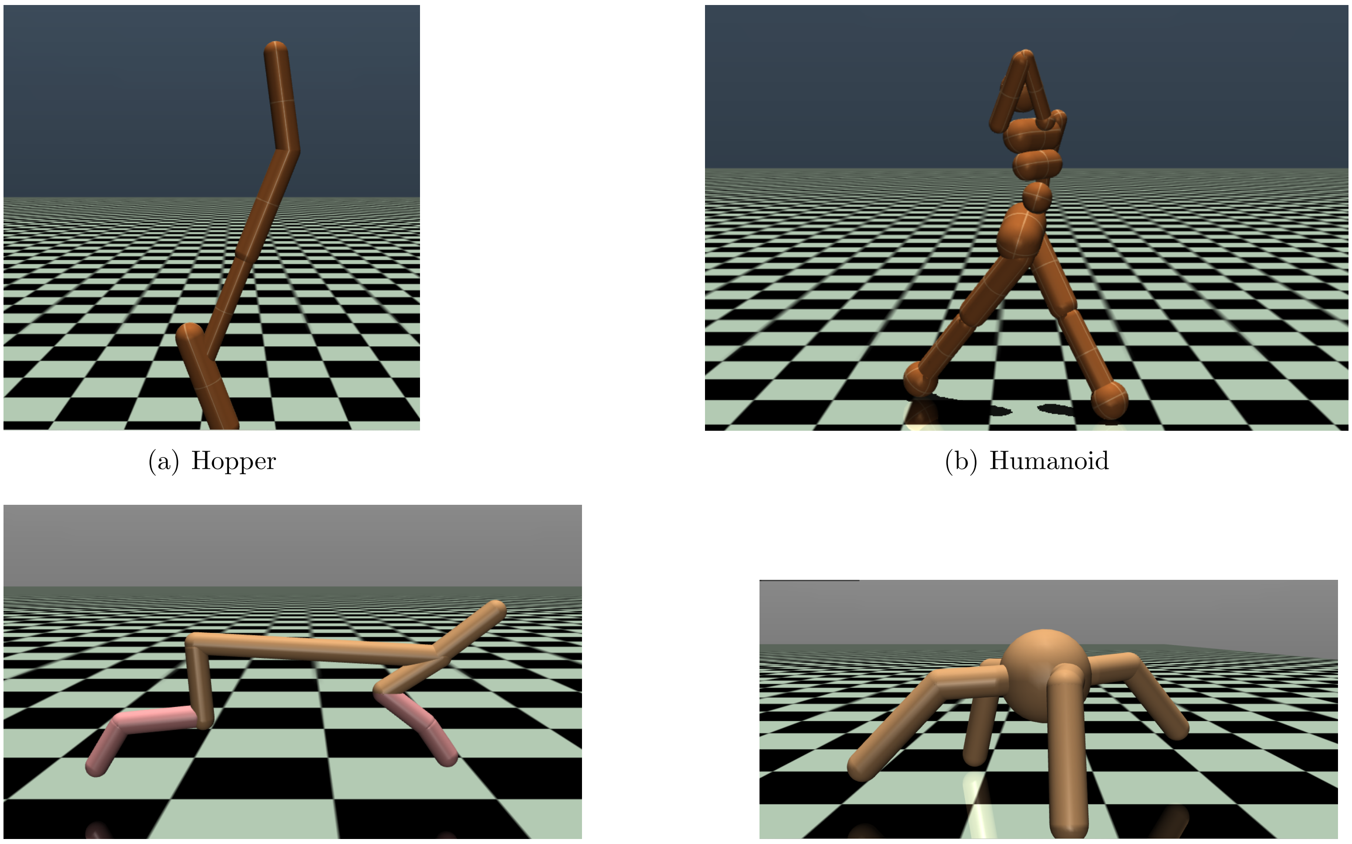

(a) Hopper
(b) Humanoid
(c) HalfCheetah
(d) Ant

Figure 2: Screen Shots of 4 MUJOCO Control

Briefly speaking, we decompose the models into basic modules (like DQN and DDPG) and advanced improving tricks (like Double DQN, Dueling DQN, Delayed DDPG, Target Smoothing DDPG, Twin DDPG etc.), which are introduced in Section 2 and compared in Section 5.
In Section 2, we mainly discuss the algorithmic and theoretical foundations of our methods. In Section 3, we mainly discussed the implementation details of our model, especially network structure and hyper-parameters, and some software and hardware concerns to ensure good performance. In Section 4, all the experimental results are visualized or listed in tables for comparisons and discussions, which are conducted in Section 5. Thanks to this project, I have a chance to experience the glory of deterministic control algorithms.

2 Methodology

In this part, the pseudo code for 2 major methods is presented. Also, 5 major tricks to improve the results are also shown.

2.1 DQN

For DQN, the algorithm is as follows. We have a replay buffer D, and 2 network of the same structure: Q and target network $\hat{Q}$.

Algorithm 1: DQN

```
1  Initialize replay memory D to capacity N
2  Initialize DQN Q with random weights $\theta$
3  Initialize target action-value function $\hat{Q}$ with weights $\theta^- \longleftarrow \theta$
4  for episode=1 to E do
5      $s_1 \longleftarrow$ Env
6      for t=1 to T do
7          $a_t \longleftarrow \epsilon - greedy(Q)$
8          $r_t, s_{t+1} \longleftarrow Env(a)$
9          $(s_t, a_t, r_t, s_{t+1}) \longrightarrow D$
10         $(s_j a_j, r_j, s_{j+1}) \longleftarrow D$
11         Set $y_j = t_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
12         descend MSE Loss: $\nabla_\theta (y_j - Q(s_{j+1}, a_j); \theta))^2$
13         For every C steps $\hat{Q} \longleftarrow Q$
```

We modify this base algorithm a lot by randomly choosing whether to write the experience to the buffer, and descend the TD loss once every 10 or 20 steps to save time. Also, for picture input, a preprocess phase is also involved, which will be elaborated in Section 3.

2.2 DDPG

By contrast, DDPG directly used a differentiable function $\mu_\phi(\cdot)$ to estimate the maxima of Q function. Thus, there is two step for DDPG update: minimizing TD error to estimate the Q function and estimating the maxima of Q function, given state s as input.

1. TD error: TD error is the Euclidean Distance between TD target and estimated Q function.

$$\min_\theta \ TDerror = ||(r_s^a + \gamma Q_{\theta'}(s', \mu_{\phi'}(s'))) - Q_\theta(s, \mu_\phi(s))||_2^2 \tag{1}$$

   Here, θ' and ϕ' are the parameters of the target networks, while θ and ϕ are the parameters of the current updating networks.

2. Maxima Searching:

$$\max_\phi \ Q_\theta(s, \mu_\phi(s)) \tag{2}$$

With this 2 objective functions in mind, we derive our pseudo code for DDPG as follows, in Algorithm 2.

Algorithm 2: DDPG Algorithm

```
1  Initialize $\theta$ and $\phi$.
2  Initialize replay buffer D to size N.
3  $\theta' \longleftarrow \theta,\ \phi' \longleftarrow \phi$
4  for $e$ = 1 to E episodes do
5      while Not Truncated or Terminated do
6          observe state s
7          take action with noise $a \longleftarrow clamp(\mu_\phi(s) + \epsilon),\ \epsilon \sim \mathcal{N}(0, \sigma^2)$
8          Put record into buffer $D \longleftarrow < s, a, r, s' >$
9          if Time to Update then
10             Update $\Delta\theta \longleftarrow \eta\nabla_\theta\ TDerror$
11             Update $\Delta\phi \longleftarrow \eta\nabla_\phi\ Q_\theta(s, \mu_\phi(s))$
12             if Time to Sync then
13                 $\theta' \longleftarrow \theta,\ \phi' \longleftarrow \phi$
```

Such differentiable function $\mu_\phi(\cdot)$ enables us to predict the maxima of Q function without quantizing the action space and conduct exhaustive search on such space. Besides, I didn't follow the soft allocation in the original paper but apply the hard synchronization just as DQN instead.

2.3 Improving Tricks

After literature survey, we borrow many tricks and ideas from the state-of-the art methods [5], [3], [9], [10], [2], [1].

2.3.1 Trick 1: Delayed Target Update

We know in DQN, we need to synchronize and update the target network once every C steps. Such design is to mitigate the oscillation and jitters of the target during training, because the target shall be stable and unchanged throughout a given training phase, which helps the training process to be more stable and converge faster. In Section 4 we will see how such delay can help DDPG train Hopper-v2.

2.3.2 Trick 2: Weight Decay

Many works have mentioned the potential overestimate of Q function [9], [1], where the huge quantity of neural network parameters and consequent over-fitting only worsen the case. In DDPG, an effective method is introduced to alleviate this issue, i.e., weight decay. With weight decay, not only over-fitting can be mitigated, but also the over-estimation is partially resolved. However, weight decay also leads to convergence issue, as we will see in the experiment part. Minimizing weight decay is equivalent to maximize a parameter prior distribution with 0 average, which is not the case for Q function network. Thus, small weight decay leads to over-estimation while big weight decay leads to divergence.

2.3.3 Trick 3: Double Q Networks

The only difference between DQN and Double DQN is the TD target.

$$y_j^{DoubleDQN} \longleftarrow r_j + \gamma\hat{Q}(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta), \theta^-) \tag{3}$$

It's said to be useful to put 2 network into TD target, because such structure can prevent the Q value from being overestimate.

2.3.4 Trick 4: Target Smoothing

Target Smoothing is also a technique employed in TD3, which is used to leverage the impact of mispredicted high peak of Q function by leveling its value and making it closer to that of adjacent states. By smoothing the target policy, the training will become much more stable.

$$\begin{gathered} a' = clip(\mu(s') + \epsilon, min, max),\ \epsilon \sim \mathcal{N}(0, \sigma^2) \\ TD\ target = Q(s', a') \end{gathered} \tag{4}$$

2.3.5 Trick 5: Delayed Policy Update

For DDPG, the convergence rate of Q function is usually slower because it takes several update of Q function to change its maxima. In other words, updating the Q network may not necessarily lead to different optimum policy, especially when the policy network has already resided in its local maxima.
In TD3 [1], the Q function is updated twice times more frequently than the policy network. I also implement this variant.

2.3.6 Trick 6: Dueling Q Networks

Due to the fully connectivity of the last layers in DQN, the Q value of different actions from the same state is not explicitly encoded, lacking entirety for any given states. Therefore, patently modeling value function of each state may help. Besides, another advantage function A(s,a) is required to compute Q(s,a) from V(s) + A(s,a). To further remove the unidentifiability in V(s) and A(s,a), one degree of freedom (dof) is removed by subtracting an average.

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

I combine Double DQN and Dueling DQN to have "Dueling DDQN" (DDDQN) for Atari Games. The results are shown in the following sections.

3 Implementation Details

Unlike what is in original DDPG and DQN. My hyper parameters differ from game to game in order to achieve better effects, due to less training frames, where the original setting in DQN requires training more than $10M$ frames and days of training. Consequently, their training time guarantees that the same set of hyper parameters will take effect on almost every game, which is not the case for us.

3.1 DQN Settings

The hyper parameters in original DQN [5] are not adopted due to their high memory and computation demands, unavailable for me. Their buffer size is set to be 10^6, while I set my buffer size to be $2 \times 10^4 - 10^5$ for 4 different games. The learning rate is 5e-4, larger than DQN since we train for less epochs. We anneal the ϵ noise from 0.6 to 0.1 within 500 episodes. Still different from original papers, our synchronization interval of target network is 100 to 500 frames, depending on games, rather than 10^4 frames due to out limited amount of training data and training time. Table 1 demonstrates our hyper parameters.

parameter	value (games-dependent)	description
B	32	size of minibatch
D	20,000-100,000	buffer size
lr	5e-4	learning rate
optimizer	Adam	NN optimizer
γ	0.99	discount factor
ϵ start	0.6	the noise at the start
ϵ end	0.1	the noise at the end
C	100-500	time between $\hat{Q} \longleftarrow Q$
skip frame	16-50	the frame skipped without training
episode number	100-500	/

Table 1: Hyper-parameters for DQN (some are game-dependent)

With respect to neural network structure, for ram games, such as VideoPingball-ramNoFrameskip-v4, Fully Connected Layers are involved, with 512, 256, 256 being the number of perceptrons for each layer, and for games with rgb pixel data, Convolutional Neural Networks are applied, special design is similar to the original work in DQN, though the number of features for each filer is different to save computation.

3.2 DDPG Settings

Just like DQN, most of our DDPG hyper parameters and network structure are just the same as the original implementation of DDPG [3], where only a few modifications are involved to save computation costs without worsening the performances. Unlike DDPG and TD3, the implementation of DDPG in this work applies hard synchronization (like DQN), rather than a soft one, i.e., $\theta_{targ} \longleftarrow \theta$ rather than $\theta_{targ} = (1-\tau)\theta_{targ} + \tau\theta$.

parameter	value	description
B	64	size of minibatch
D	10^5	buffer size
actor-lr	env-dependent	learning rate
critic-lr	env-dependent	learning rate
optimizer	Adam	NN optimizer
γ	0.99	discount factor
noise	$\mathcal{N}(0, \sigma^2)$	noise distribution
σ start	0.35-0.45	standard deviation (start)
σ end	0.1-0.15	standard deviation (end)
sync	hard synchronization	hard synchronization.
C	20-200	time steps between $\hat{Q} \longleftarrow Q$
skip frame	1-4	the frame skipped without training
episode number	3000-5000	/

Table 2: Hyper-parameters for DDPG

With respect to neural network structure, Fully Connected Layers are involved, with 400, 300 being the number of perceptrons for each layer for both actor and critic.

4 Experimental Results

4.1 Base Experiments

The base experiments are directly conducted with the base implementations of DQN and DDPG without any tricks at all.
Human Rewards here refers to myself. I, a beginner, learn 10 minutes of playing given Atari Games before testing 2 times on it and record my average rewards as the human reward. For Atari Games, max steps is set to 10^4 to let the game to finish.

env	method	reward (max steps for MUJOCO: 1000)	human reward	agent-human
Ant	DDPG	1468.96	/	/
HalfCheetah	DDPG	933.46	/	/
VideoPingball	DQN	7468.4	5210	143%

Table 3: Base Results

Here, only part of all environments are listed out since agents in other environment aren't trained well.
The results of Ant and HalfCheetah are illustrated in Figure 2.

4.2 Trick Effects

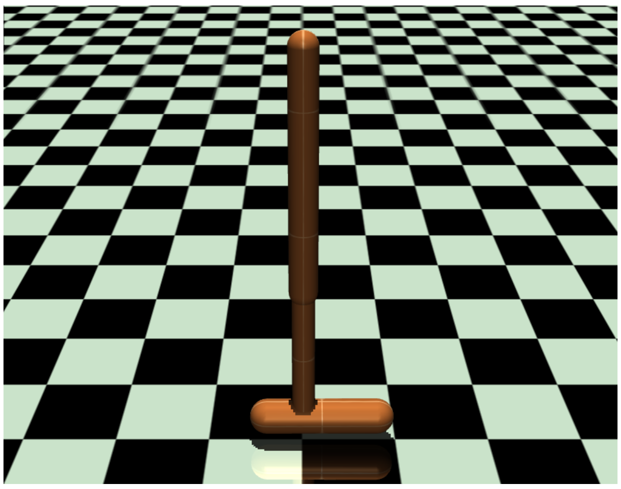
4.2.1 The effect of delayed target update

We apply delay our target update on Hopper-v2 and achieve astounding effects.

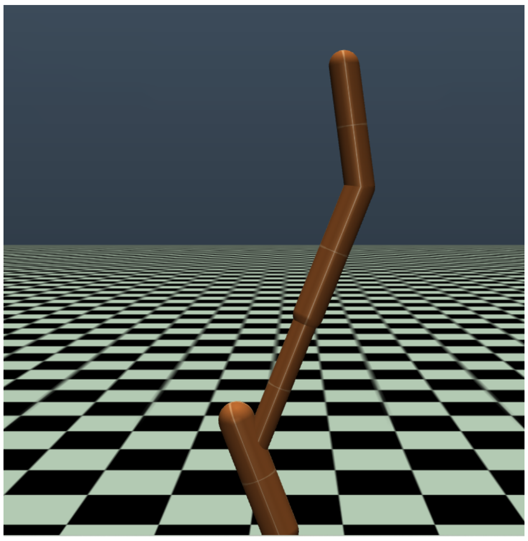
method	env	update interval	avg rewards	survive steps (max: 1000)
DDPG	Hopper-v2	10	5.82	8
DDPG	Hopper-v2	50	**3570.95**	**1000**

Table 4: The Effect of Delayed Target Update

The visualization of the results can also be illustrated directly.



(a) Hopper (C=10)



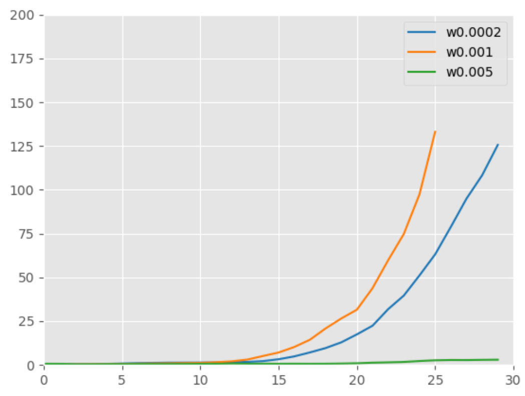
(b) Hopper (C=50)

Figure 3: Hopper Comparison for Delayed Target Update

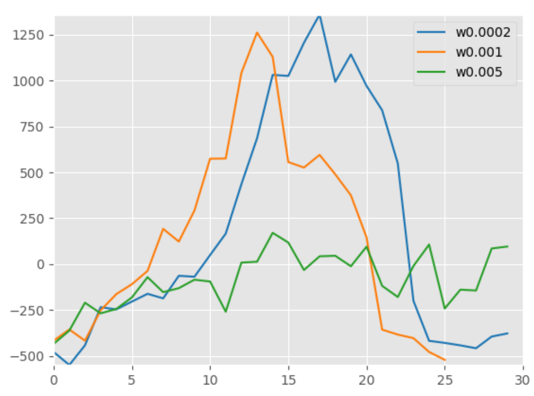
The Hopper trained with C=10 stand steadily before collapse, while C=50 hops smoothly.

4.2.2 The effect of weight decay

As is discussed before, weight decay can prevent over-fitting as well as over-estimation of Q function. The effect of weight decay on HalfCheetah is intriguing.
The following figures show the loss and reward curves of the agent under different weight decay.



(a) loss curve



(b) reward curve

Figure 4: loss and reward for different weight decay factor

It demonstrates clearly that a small weight decay may result in the divergence of Q function, while a weight decay may dominant the whole loss and thus also force the whole network to diverge. Therefore, only for w = 0.005, the loss is stable and thus the training process can thus be carried out, while for w=0.001 and w=0.0002, the reward quickly hit a peak and diverges from then on.

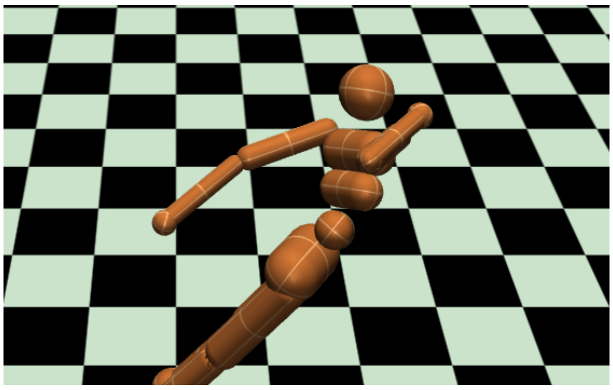
4.2.3 The effect of target smoothing and delayed policy update

Humanoid-v2 is the most difficult environment among all 4 MUJOCO environments because of its the most complicated dynamics and its biggest action and observation space. Without target smoothing and delayed policy update, the agent can hardly learn anything. By contrast, with the help of target smoothing and delayed policy updates, the agent slowly learn moving ahead by throwing its left leg ahead repeatedly.

method	env	trick	avg rewards	survive steps (max: 1000)
DDPG	HalfCheetah-v2	/	933.46	1000
Delayed DDPG	HalfCheetah-v2	target noise=0.01	**7097.35**	1000
DDPG	Humanoid-v2	/	345.68	76
Delayed DDPG	Humanoid-v2	target noise=0.01	**3614.43**	**800+**

Table 5: The Effect of Delayed Target Smoothing

The figures below illustrated the effects of target smoothing and delayed policy update, which significantly helped the training to converge.


(a) Humanoid trained without tricks

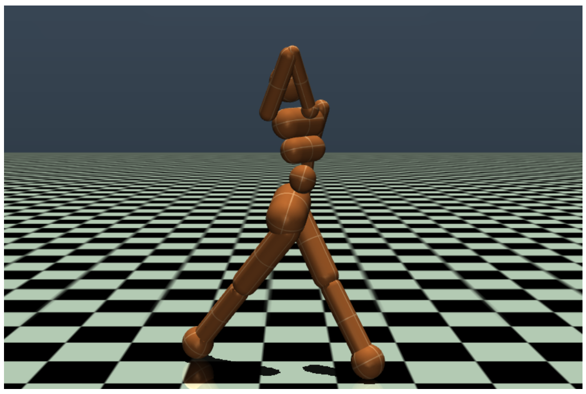
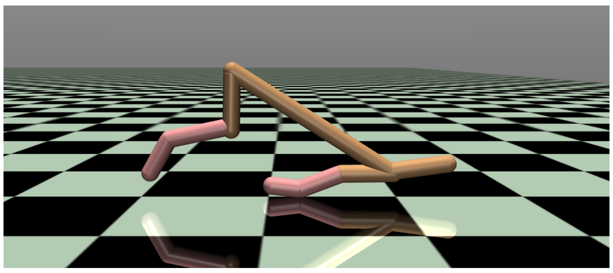

(b) Humanoid trained with tricks

Figure 5: Humanoid-v2 w/o target smoothing and delayed policy update

HalfCheetah also stands up after the trick.


(a) HalfCheetah trained without tricks

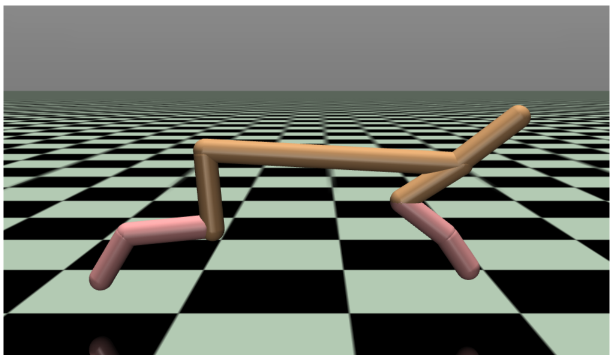

(b) HalfCheetah trained with tricks

Figure 6: HalfCheetah-v2 w/o target smoothing and delayed policy update

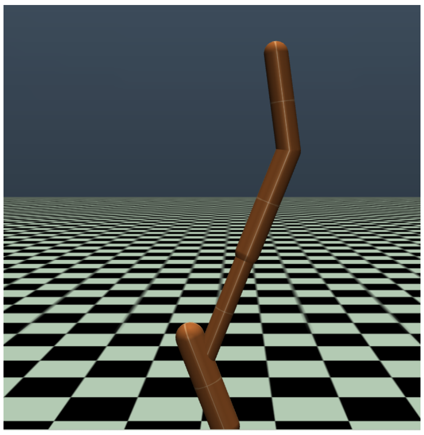
4.3 Final Results

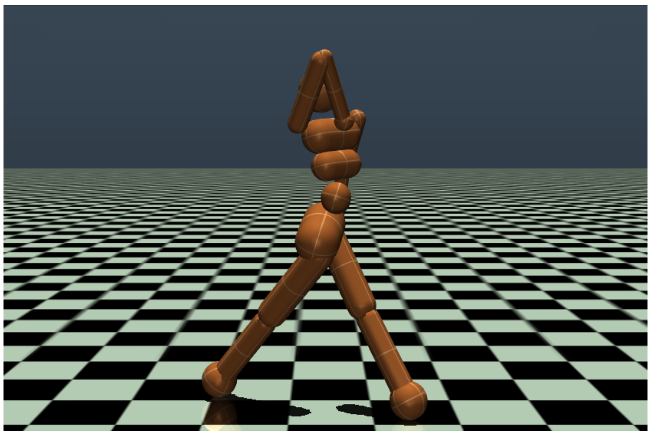
The final results of our project are given by the best outcome from all these tricks. **(The max steps for MUJOCO is 1000)** The agent didn't learn anything on Pong and Breakout due to time limitation.

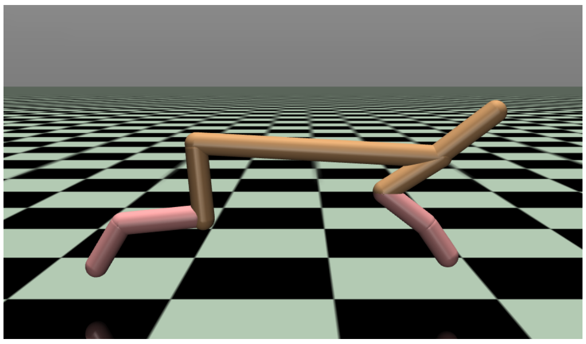
env	method+trick	reward	human reward	agent v.s. human
Ant	DDPG	1468.96	/	/
HalfCheetah	DDPG+weight decay	7097.35	/	/
Hopper	DDPG+delayed target update	3570.95	/	/
Humanoid	Delayed DDPG+smoothing	3614.43	/	/
Boxing	DDQN	-4	5	human better
Breakout	Dueling DDQN	3 (×)	30	/
Pong	Dueling DDQN	-21 (×)	-4.5	/
VideoPingball	DDQN	7468.4	5210	agent better

Table 6: Best Results on All Environments

The results are illustrated below in Figure 7 and Figure 8. All demo videos are available in my Github repository: https://github.com/huangzhengxiang/RL_Project.git.


(a) Hopper


(b) Humanoid


(c) HalfCheetah

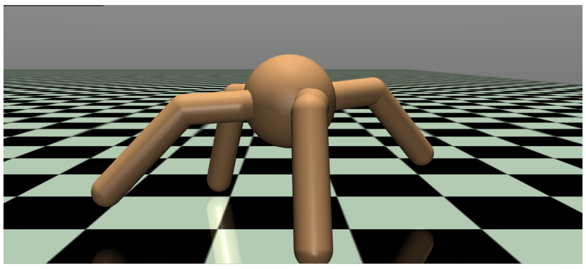
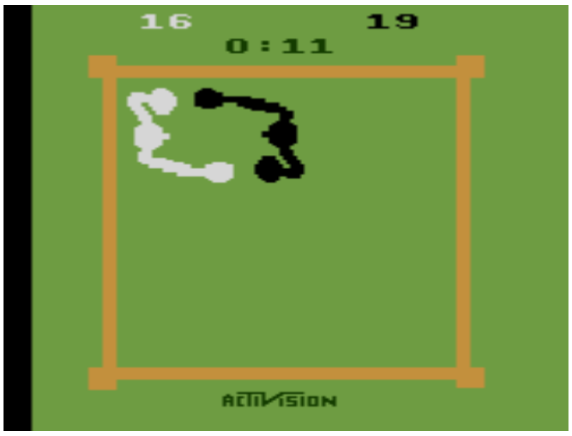
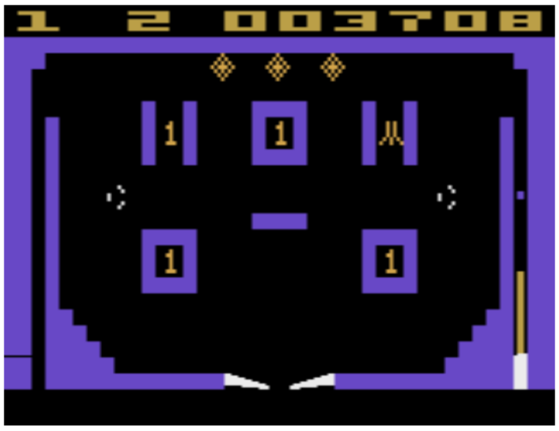

(d) Ant

Figure 7: Illustration of Best MUJOCO Results

It's a pity that our agents didn't tower over human due to the lack of sufficient training time and computation resources. Fortunately, the visualizations indicate that they surely have learned something useful after all, as is shown in Figure 8. In order to illustrate the agent indeed learn from the games, I choose 2 encouraging screen shots from the games: dog fight in Boxing, 2 players punching each other at the same time, and pop-up in VideoPinball, popping up the ball again after loosing 1 life. These actions indicate that the agent is not choosing some random actions, but taking some meaningful and valuable ones to gain rewards in games.



(a) dog-fight in Boxing



(b) pop-up in VideoPinball

Figure 8: Illustration of Best Atari Results

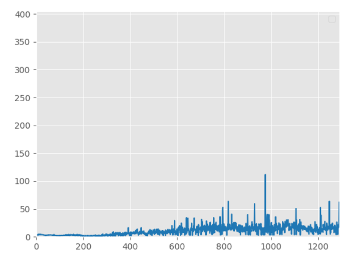
5 Discussion

5.1 The Effect of Deterministic Control

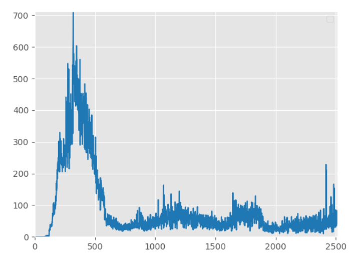
For any RL algorithms with a probabilistic policy, the distribution of the policy function shall be first modeled as a prior initially. Searching for a optimal policy distribution is hard, and thus most probabilistic models model such prior policy function distribution as Multinomial distribution, Gaussian distribution, Dirichlet distribution, or other simple and differentiable distribution. However, such effects may be an ineffective attempt, because when multinomial and Gaussian is adopted, the probabilistic model degrades to a nearly deterministic one.
If the true distribution is highly non-Gaussian, then fitting it with a prior distribution is also hard. Therefore, inherently, probabilistic actor-critic models can perform no better than their deterministic counterparts, while requiring more human labor and design. As far as the expressibility is concerned, polynomial-time deterministic control is somehow equivalent to their probabilistic counterparts.

5.2 Oscillation

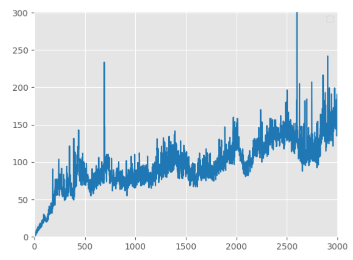
Most of the state-of-the-art methods pay close attention to the oscillation or variation during training process. During the experiments in this project, such variation is also observed.



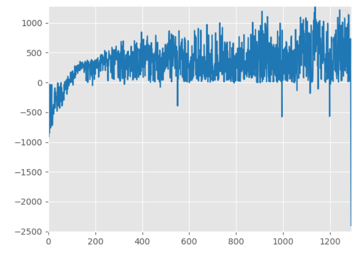
(a) Ant loss curve



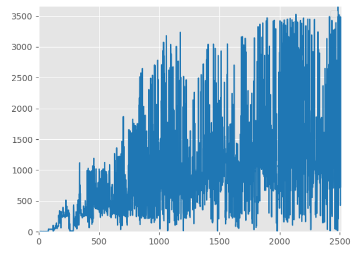
(b) Hopper loss curve



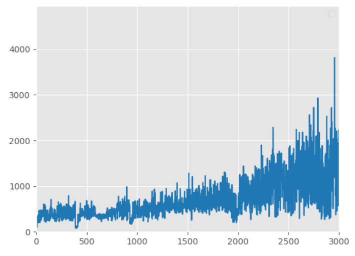
(c) Humanoid loss curve



(d) Ant reward curve



(e) Hopper reward curve



(f) Humanoid reward curve

Figure 9: Oscillation during Training

As is shown in Figure 9, both the loss and reward oscillates throughout training, due in part to the exploration noise and consequent variation. Because the exploration is essential, such oscillation is the inherent characteristics for RL training. Therefore, methods to eliminate noises other than exploration can alleviate the overall variance and lead to a probable convergence after the noise is annealed to 0.

6 Acknowledgement

This is the final project for the course. I've done my best to finish the task and learn from it. Finally, he best possible results are accomplished on MUJOCO, and 2 of 4 Atari games are trained quite well. I'm certain that I've indeed learned a lot.
A lot thanks to our kind instructor Zhou JN, and our responsible TA. You've delivered us an awesome lesson.
I also express my gratitude to my classmates, Zhang JM, Dong HJ, Liao WX, Liu K, Guo JP, Wu QY, and all my friends for your discussions with me and your help.

References

[1] Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. In *INTERNATIONAL CONFERENCE ON MACHINE LEARNING, VOL 80* (2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*. 35th International Conference on Machine Learning (ICML), Stockholm, SWEDEN, JUL 10-15, 2018.

[2] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *THIRTY-SECOND AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE / THIRTIETH INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE CONFERENCE / EIGHTH AAAI SYMPOSIUM ON EDUCATIONAL ADVANCES IN ARTIFICIAL INTELLIGENCE* (2018), AAAI Conference on Artificial Intelligence, AAAI, pp. 3215–3222. 32nd AAAI Conference on Artificial Intelligence / 30th Innovative Applications of Artificial Intelligence Conference / 8th AAAI Symposium on Educational Advances in Artificial Intelligence, New Orleans, LA, FEB 02-07, 2018.

[3] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Yuval Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning, 2015.

[4] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., and Lillicrap, T. P. Asynchronous methods for deep reinforcement learning. In *INTERNATIONAL CONFERENCE ON MACHINE LEARNING, VOL 48* (2016), M. Balcan and K. Weinberger, Eds., vol. 48 of *Proceedings of Machine Learning Research*. 33rd International Conference on Machine Learning, New York, NY, JUN 20-22, 2016.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *NATURE 518*, 7540 (FEB 26 2015), 529–533.

[6] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms. In *INTERNATIONAL CONFERENCE ON MACHINE LEARNING* (2014), pp. 387–395.

[7] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[8] van Hasselt, H. Double q-learning. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 23 (NIPS 2010)* (2010), vol. 23, p. 2613–2621.

[9] van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning, 2015.

[10] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. Dueling network architectures for deep reinforcement learning, 2016.