REPORT OF CS489 FINAL PROJECT

Hongzhou Liu
517030910214
deanlhz@sjtu.edu.cn

1 INTRODUCTION

In this project, I implemented some value-based and policy-based RL algorithm and trained them on different environments based on `gym`. The key required packages are

- PyTorch
- gym
- mujoco-py (with MuJoCo)

As we learned in the class, value-based RL methods strive to fit action value function or state value function and control agent in off-policy or on-policy ways. Deep Q-Learning is a groudbreaking work that introduces Deep Neural Networks to Q-learning algorithm. However, policy-based method mainly focus on fitting a policy directly. It is achieved by adjusting policy function by policy gradient method. However, the original policy gradient method suffers from high variance, and then Actor-Critic method was proposed. Also, the Actor-Critic method is sample inefficient and it used on-policy. Thus, off-policy algorithm like DDPG is introduced.

2 METHODS

2.1 DEEP Q-LEARNING

The original table-based Q-learning method can be sufficiently applicable to the environment where the all achievable states can be managed and stored in RAM. However, the environment where the number of states overwhelms the capacity of contemporary computers, for example, the Atari games, the original approach is not very applicable. Thus, the Q-table in the original method is replaced by a neural network called Q-network. And such method is called Deep Q-learning, proposed by Mnih et al. (2015).

The two key points in Deep Q-learning is fixed target net and replay buffer. We use the target net to predict target value as label. And we use replay buffer to memorize the history and it can be sampled from when training the actual Q-network. The DQN algorithm is shown in Algorithm 1

The original DQN takes the maximum overestimated values as such is implicitly taking the estimate of the maximum value. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping learning estimates from estimates such overestimation can be problematic. Thus, Hasselt et al. (2016) proposed Double DQN to tackle this over-estimate problem. The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates. As we can see in DQN, we calculate target value by

$$r_j + \gamma \max_{a'} \hat{Q}\left(s_{j+1}, a'; \hat{\boldsymbol{\theta}}\right)$$

so that the max operator uses the same values to both select and evaluate an action. Thus, it's more likely to select over-estimated values, and results in over-optimistic value estimates. Then, in Double DQN, above problem is solved by changing the evaluation of target value in the algorithm into

$$r_j + \gamma \hat{Q}\left(s_{j+1}, \arg\max_{a'} Q(s_{j+1}, a'; \boldsymbol{\theta}); \hat{\boldsymbol{\theta}}\right)$$

Algorithm 1: DQN

Initialize replay buffer D
Initialize action-value function Q with random weight $\boldsymbol{\theta}$
Initialize target action-value function $\hat{Q}$ with $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}$
for *episode = 1 to* M **do**
  Initialize the episode and acquire the initial state s_1
  for t *= 1 to* T **do**
    Following ϵ-greedy policy, select action

$$a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q\left(s_t, a; \boldsymbol{\theta}\right) & \text{otherwise} \end{cases}$$

    Obtain reward r_t and next state s_{t+1} by executing the action a_t in the environment
    Store transition (s_t, a_t, r_t, s_{t+1}) in the buffer D
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from buffer D
    Calculate target value by

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

    Perform a gradient descent on $(y_j - Q(s_j, a_j; \boldsymbol{\theta}))$ w.r.t the network parameter $\boldsymbol{\theta}$
    Every C step reset $\hat{Q} = Q$ by $\hat{\boldsymbol{\theta}} = \boldsymbol{\theta}$

Furthermore, in order to converge faster than original DQN, Wang et al. (2016) proposed the dueling network architecture to estimate state value function $V(s)$ and associated advantage function $A(s, a)$, and then combine them to estimate action value function $Q(s, a)$. In the DQN architecture of Atari tasks, a CNN layer is followed by a fully connected (FC) layer. In dueling architecture, a CNN layer is followed by two streams of FC layers, to estimate value function and advantage function separately; then the two streams are combined to estimate action value function. In this paper, the author proposed the following way to combine $V(s)$ and $A(s, a)$ to obtain $Q(s, a)$:

$$Q(s, a; \boldsymbol{\theta}, \alpha, \beta) = V(s; \boldsymbol{\theta}, \beta) + \left(A(s, a; \boldsymbol{\theta}, \alpha) - \frac{a}{|\mathcal{A}|} A\left(s, a'; \boldsymbol{\theta}, \alpha\right) \right)$$

where α and β are parameters of the two streams of FC layers, $|\mathcal{A}|$ is the size of action space. The architectures of DQN and Dueling DQN are illustrated in Figure 1.

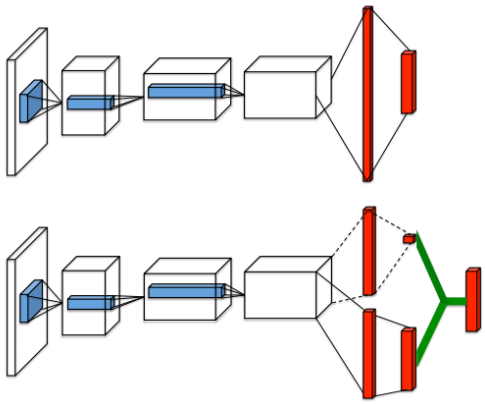


Figure 1: Network Architectures

2.2 Asynchronous Advantage Actor-Critic

A3C is a policy-based RL algorithm proposed by Mnih et al. (2016). We know that a policy maps state to action, and policy optimization is to find an optimal mapping. Compared with original policy gradient method, A3C benefits from Actor-Critic. An actor-critic algorithm learns both a policy and a state-value function, and the value function is used for bootstrapping, i.e., updating a state from subsequent estimates, to reduce variance and accelerate learning. In A3C, parallel actors employ different exploration policies to stabilize training, so that experience replay is not utilized. Different from most deep learning algorithms, asynchronous methods can run on a single multi-core CPU. The algorithm for each actor-learner thread is shown in Algorithm 2.

Algorithm 2: A3C, actor-learner thread

Global shared parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_v$, thread-specific parameter vectors $\boldsymbol{\theta}'$ and $\boldsymbol{\theta}'_v$
Global shared counter $T = 0$, T_{max}
Initialize thread step counter $t \leftarrow 1$
for $T \leq T_{max}$ **do**
    Reset gradients, $d\boldsymbol{\theta} \leftarrow 0$ and $d\boldsymbol{\theta}_v \leftarrow 0$
    Synchronize thread-specific parameters $\boldsymbol{\theta}' = \boldsymbol{\theta}$ and $\boldsymbol{\theta}'_v = \boldsymbol{\theta}_v$
    Set $t_{start} = t$ and get state s_t
    for s_t *not terminal and* $t - t_{start} \leq t_{max}$ **do**
        Perform a_t according to policy $\pi(a_t|s_t;\boldsymbol{\theta}')$
        Receive reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V\left(s_t, \boldsymbol{\theta}'_v\right) & \text{otherwise} \end{cases}$
    for $i \in \{t-1, \cdots, t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients w.r.t. $\boldsymbol{\theta}'$: $d\boldsymbol{\theta} \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi\left(a_i \mid s_i;\boldsymbol{\theta}'\right)\left(R - V\left(s_i;\boldsymbol{\theta}'_v\right)\right)$
        Accumulate gradients w.r.t. $\boldsymbol{\theta}'_v$: $d\boldsymbol{\theta}_v \leftarrow d\boldsymbol{\theta}_v + \nabla_{\boldsymbol{\theta}'_v}\left(R - V\left(s_i;\boldsymbol{\theta}'_v\right)\right)^2$
    Perform asynchronous update of $\boldsymbol{\theta}$ using $d\boldsymbol{\theta}$, and of $\boldsymbol{\theta}_v$ using $d\boldsymbol{\theta}_v$

As seen above, the A3C maintains a policy $\pi(a_t|s_t;\boldsymbol{\theta})$ and an estimate of the value function $V(s_t;\boldsymbol{\theta}_v)$, being updated with n-step returns in the forward view, after every t_{max} actions or reaching a terminal state, similar to using minibatches. The gradient can be seen as $\nabla_{\boldsymbol{\theta}'} \log \pi\left(a_t \mid s_t;\boldsymbol{\theta}'\right) A\left(s_t, a_t;\boldsymbol{\theta},\boldsymbol{\theta}_v\right)$, where $A\left(s_t, a_t;\boldsymbol{\theta},\boldsymbol{\theta}_v\right) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V\left(s_{t+k};\boldsymbol{\theta}_v\right) - V(s_t;\boldsymbol{\theta}_v)$ is an estimate of the advantage function, with k upbounded by t_{max}.

2.3 Deep Deterministic Policy Gradient

Policies are usually stochastic. However, Silver et al. (2014) and Lillicrap et al. (2016) proposed deterministic policy gradient (DPG) for efficient estimation of policy gradients. Silver et al. (2014) introduced the deterministic policy gradient (DPG) algorithm for RL problems with continuous action spaces. The deterministic policy gradient is the expected gradient of the action-value function, which integrates over the state space; whereas in the stochastic case, the policy gradient integrates over both state and action spaces. Consequently, the deterministic policy gradient can be estimated more efficiently than the stochastic policy gradient. The authors introduced an off-policy actor-critic algorithm to learn a deterministic target policy from an exploratory behaviour policy, and to ensure unbiased policy gradient with the compatible function approximation for deterministic policy gradients.

Lillicrap et al. (2016) proposed an actor-critic, model-free, deep deterministic policy gradient (DDPG) algorithm in continuous action spaces, by extending DQN Mnih et al. (2015) and DPG Silver et al. (2014). With actor-critic as in DPG, DDPG avoids the optimization of action at every time step to obtain a greedy policy as in Q-learning, which will make it infeasible in complex action spaces with large, unconstrained function approximators like deep neural networks. To make the

learning stable and robust, similar to DQN, DDPQ deploys experience replay and an idea similar to target network, soft target, which, rather than copying the weights directly as in DQN, updates the soft target network weights $\boldsymbol{\theta'}$ slowly to track the learned networks weights $\boldsymbol{\theta}$: $\boldsymbol{\theta'} \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta'}$ with $\tau \ll 1$. The authors adapted batch normalization to handle the issue that the different components of the observation with different physical units. As an off-policy algorithm, DDPG learns an actor policy from experiences from an exploration policy by adding noise sampled from a noise process to the actor policy. Algorithm 3 shows the detail of DDPG.

Algorithm 3: DDPG

Randomly Initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for *episode = 1 to* M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for *t = 1 to* T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

To choose a suitable random process $\mathcal{N}$, in Lillicrap et al. (2016), the author utilized Ornstein-Uhlenbeck process proposed by Uhlenbeck & Ornstein (1930) to generate tempoprally correlated exploration for exploration efficiency in physical control problems with inertia. Furthermore, above random process may sometimes lead to local minimum problem. In Plappert et al. (2017), the author proposed parameter space noise for exploration. Such method add noise directly to the agent's parameters, which can lead to more consistent exploration and a richer set of behaviors. The three key points of this method are

- State-dependent exploration: We can still utilize Ornstein-Uhlenbeck process to generate a random process.
- Perturbing deep neural networks: Noise is added to the parameters of neural networks, here in DDPG, we add noise to the actor network.
- Adaptive noise scaling: The parameter noise requires us to pick a suitable scale σ, which is hard to choose. With adaptive noise scaling, such problem will be solved.

3 EXPERIMENTS

In this part, I trained and tested Dueling Double DQN on the following Atari Games Environment:

- BreakoutNoFrameskip-v4
- PongNoFrameskip-v4

Also, I trained and tested A3C, DDPG and DDPG with parameter space noise on the following MuJoCo Continuous Control Environment:

- Hopper-v2
- HalfCheetah-v2

3.1 Dueling Double DQN

3.1.1 Experiment Setting

I followed the training process in Dhariwal et al. (2017) and Mnih et al. (2015). The observation space of Atari Games Environment is $210 \times 160 \times 3$ here. We need to process the original observations in order to save memory and acquire better performance. First, we need to resize the original input to 84×84 and convert it from the RGB space to grayscale. Also, we have to store the observations into the replay buffer in the type of uint8 to save memory and convert them back to float when sampled from the buffer. Also, similar to Dhariwal et al. (2017), I stacked 4 consecutive frames together as a state in order to encode the movement of the game. I also performed frameskip and reward clipping to stabilize the training process. For BreakoutNoFrameskip-v4, an episode consists of 5 trials, I split it into 5 episodes to keep the same as what Mnih et al. (2015) did in their experiments. The hyper-parameter setting is shown in Table 1. For the detailed implementation and network structure, please refer to the source code. I trained BreakoutNoFrameskip-v4 for 50000 episodes and PongNoFrameskip-v4 for 5000 episodes on a computer with AMD Ryzen 3700X, 128GB Memory and Nvidia GTX 1080Ti. The maximum memory consumption is about 77GB and the training will cost about 10 hours per task.

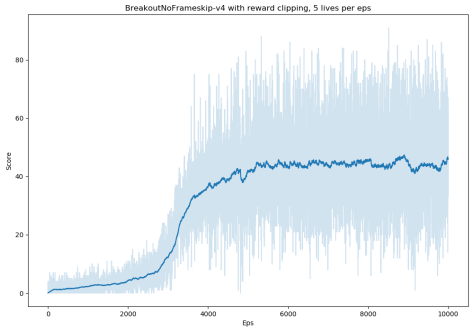
Table 1: Hyper-parameters of Dueling Double DQN

Hyper-parameter	Value
Replay Buffer Size	1000000
γ	0.99
ϵ_0	1.0
ϵ_t	0.1
ϵ decay rate	1000000
Batch Size	32
Learning rate η	0.0002
Stride to perform gradient descent	4
Stride to update target network C	2000

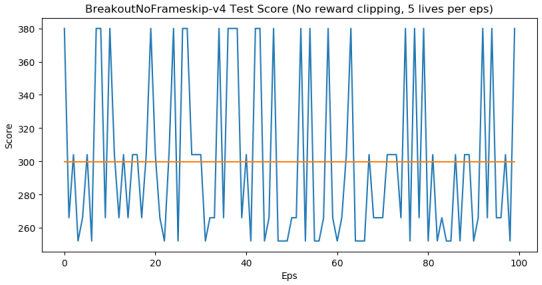
3.1.2 Result

The training and testing result is shown in Figure 2 and 3. As you can see, in BreakoutNoFrameskip-v4 environment, the network starts to converge after 4000 episodes(5 lives per episodes). I can't get a good result with small buffer size, because DQN suffers from cascade forgetting problem with small buffer size when the state number is large, for example, here in Atari games. I have to hire a machine with quite enormous memory to prevent such problem. Thus, as you can see, the result is quite good. In the testing phase, I cancelled reward clipping, and the Dueling Double DQN can reach average score of 300 and highest score of 380. However, compared with the result in Mnih et al. (2015), which is 401.2($\pm$24.6), it's still a little bit low. It is still due to the lack of training. I just trained the model for 10000 episodes and the model must stuck in some local minimum. As you can see in the rendering environment in Figure 4, when the episode is ended with score of 380, there's still some blocks that are not broken. Also, when I investigate the rendering when testing, in such case, the slab choose to stuck at the right bottom. It shows that the model is stuck in the sub-optimal solution. The way to tackle this problem and reach a higher score is to allow more exploration and train the model for more episodes.

In PongNoFrameskip-v4, the network starts converging after 1000 episodes which is quite fast, and it soon gets stabled with a score around 15. The testing result is also quite surprising. It keeps the score at 21 which is higher than the result shown in Mnih et al. (2015), which is 18.9($\pm$1.3). So, comparing



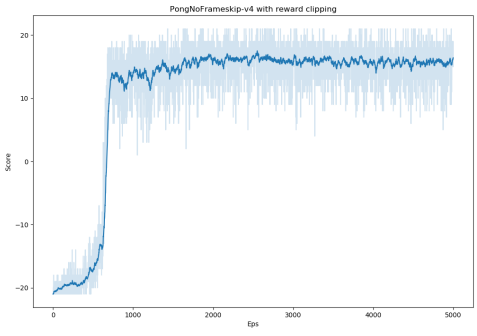
(a) Train



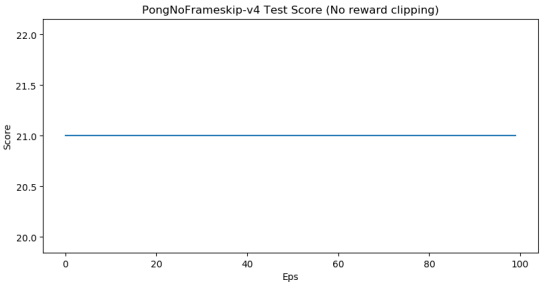
(b) Test

Figure 2: BreakoutNoFrameskip-v4

the performance of Dueling Double DQN on the two different environments, we can conclude that an RL algorithm should be fine tuned on different environments to reach a better performance. Also, a quantity of computing resources are significant to train a good model. However, we should also optimize the usage of computing resources and compute in a more efficient way.

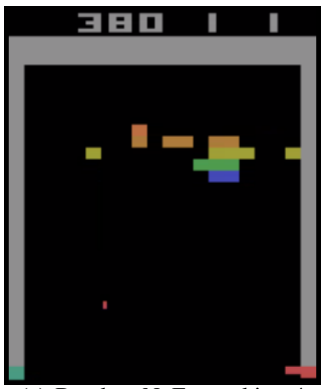


(a) Train

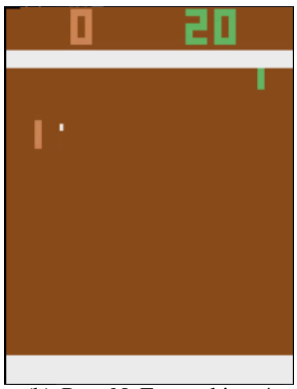


(b) Test

Figure 3: PongNoFrameskip-v4



(a) BreakoutNoFrameskip-v4



(b) PongNoFrameskip-v4

Figure 4: Test

3.2 A3C

3.2.1 Experiment Setting

In this part, I trained A3C on two MuJoCo Continuous Control Environments. The hyper-parameter settings is shown in Table 2. I trained both Hopper-v2 and HalfCheetah-v2 for 10000 episodes.

Table 2: Hyper-parameters of A3C

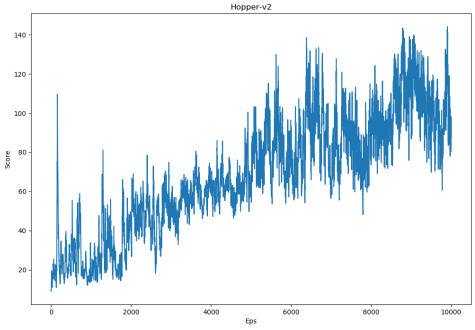
Hyper-parameter	Value
Thread number	8
γ	0.9
Learning rate η	0.00001
Stride to update global network	10

Specially, for HalfCheetah-v2, I set the the number of maximum steps as 1000. For the detailed network architectures, you can refer to the source code. Those tasks are trained on the HPC provided by the course.

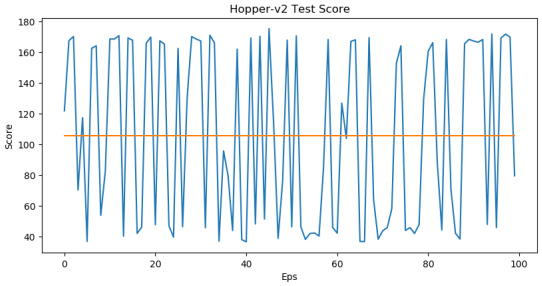
3.2.2 Result

Though A3C is quite efficient to train and buffer-free, itis not a good choice to train both Hopper-v2 and HalfCheetah-v2. In the previous assignment, we've already seen the its sensitivity to the hyper-parameters and its unstable results. Here, as you can see in Figure 5, when training Hopper-v2, the score is rising with fluctuation and reaches about 140 at the end. When testing, the score also fluctuates between 40 and 170.

When training HalfCheetah-v2, the result is even worse. In Figure 6, it starts with a sequence of quite bad actions, acquiring scores with -40000. There's a 0 plateau after about 500 episodes and it soon dropped back to negative scores. The testing score is also not good. It shows that the performance of A3C between different environments is quite large. A key difference is that, an episode in Hopper-v2 will be terminated when the "leg" in the environment acts abnormally while the "cheetah" in HalfCheetah-v2 will never terminated instead of terminating it by hand. The training results will be affected by such difference because when the "leg" acts bad the episode will be terminated and the agent will "detect" this, then affects the update of networks. And in HalfCheetah, such bad actions will not affect the training process. Above all, we can conclude that A3C is a bad choice to train in MuJoCo Continuous Control Environments.

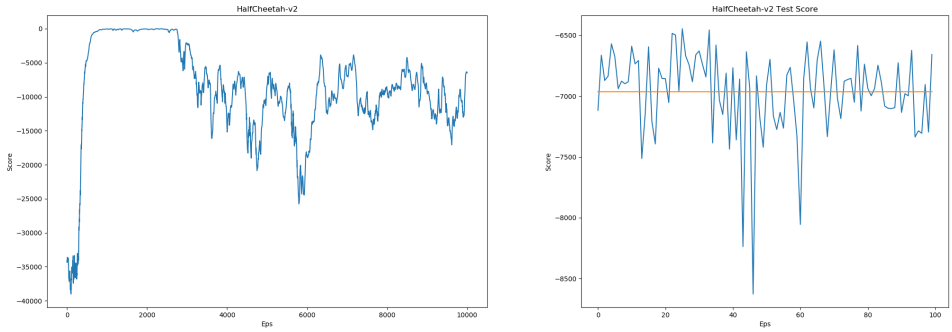


(a) Train



(b) Test

Figure 5: Hopper-v2



(a) Train (b) Test

Figure 6: HalfCheetah-v2

3.3 DDPG

3.3.1 Experiment Setting

In this section, I trained DDPG and it's improvement on Hopper-v2 and HalfCheetah-v2. The settings of hyper-parameter is in 3. Specially, for the parameter space noise method, the adaptation

Table 3: Hyper-parameters of DDPG

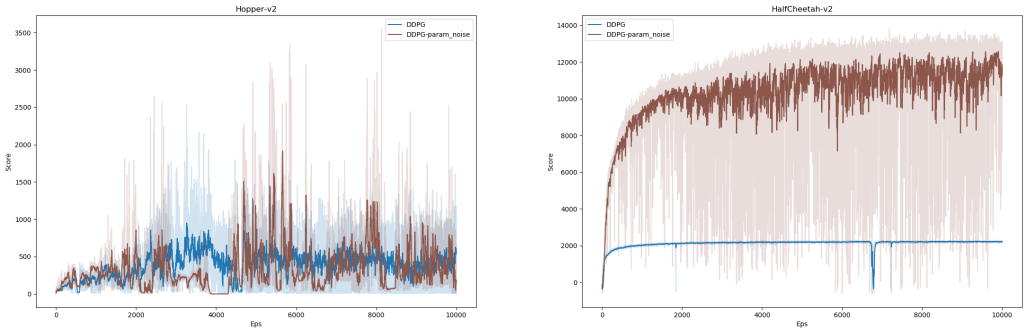
Hyper-parameter	Value
Replay Buffer Size	1000000
γ	0.99
Batch Size	64
Learning rate of actor η_a	0.0001
Learning rate of critic η_c	0.001
Soft update factor τ	0.001
Stride to update target network C	30

coefficient is set to 1.01. For other details, please refer to the source code. As in A3C, I trained both Hopper-v2 and HalfCheetah-v2 for 10000 episodes. Specially, for HalfCheetah-v2, I set the the number of maximum steps as 1000. Also I trained them on the HPC provided by the course.

3.3.2 Result

In order to achieve a good result in Hopper-v2 and HalfCheetah-v2, I trained DDPG and DDPG with parameter space noise. The training results of both methods on the two environments is are shown in Figure 7. This time, the DDPG and its improvement worked quite good on HalfCheetah-v2 and not so good on Hopper-v2. The score of HalfCheetah-v2 converged in less than 500 episodes using DDPG and about 2000 when using its improvement. However, you can observe that the performance of DDPG with parameter space noise (PSN) is 6 times higher than the original DDPG no matter when training or testing. By observing the result of training Hopper-v2 in Figure 7, you will find that the model does not converge and fluctuates a lot in both methods. Though DDPG with PSN will sometimes hit a higher score, it seems more unstable than the original method.

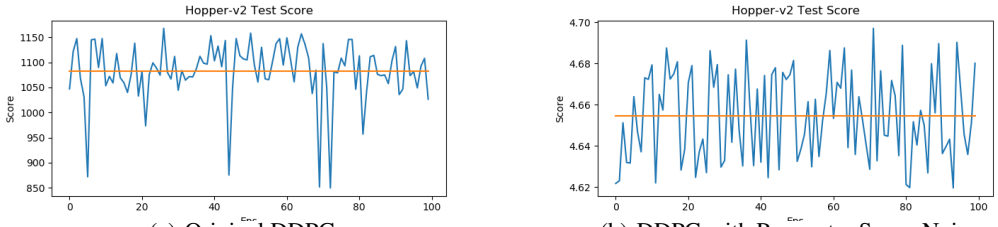
In the testing phase, you will find in Figure 9 the DDPG and its improvement also worked well and they kept the "6 times" relationship. It shows the power of parameter space noise. Combining the rendering result of testing in Figure 10, you can see the original DDPG stuck in a sub-optimal solution which makes the "cheetah" running on its back while the DDPG with PSN finds the global optimal that makes the "cheetah" run in a normal posture and in a fast speed. It also reminds us the power of parameter space noise compared with action space noise, because the former allows more and nicer explorations to prevent local optimal solution.



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

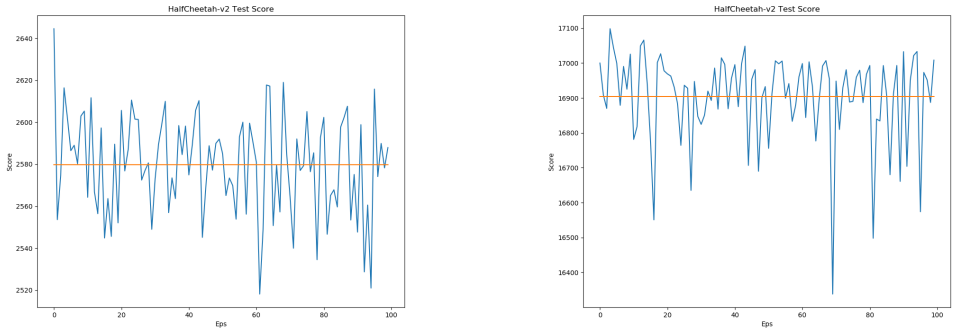
Figure 7: Train DDPG

However, in Figure 8, the advantage of DDPG with PSN disappeared, instead the original DDPG worked well. It's none about the advantage or disadvantage of algorithm, it's just because the model is not converged. And when the training of DDPG finished, we fortunately saved a model that can work in some ways. And when the training of DDPG with PSN finished, we didn't save a good model. It proves in another way that both DDPG and its improvement are not converged. The huge difference of performance between Hopper-v2 and HalfCheetah-v2 is also related to the different property of environments as we mentioned in the A3C part. Due to the nonstop property, HalfCheetah-v2 is suitable to train with DDPG while Hopper-v2 is not suitable and even unstable. It shows that the different environment will affect the algorithm somehow.



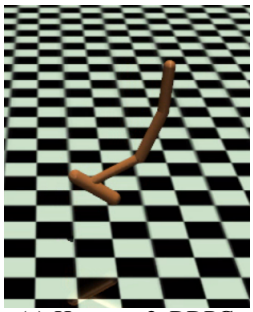
(a) Original DDPG (b) DDPG with Parameter Space Noise

Figure 8: Test Hopper-v2

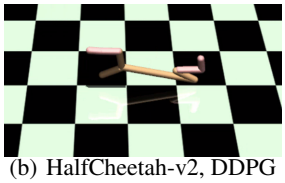


(a) Original DDPG (b) DDPG with Parameter Space Noise

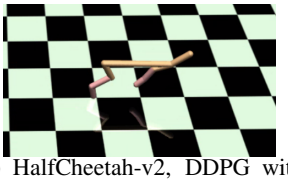
Figure 9: Test HalfCheetah-v2



(a) Hopper-v2, DDPG



(b) HalfCheetah-v2, DDPG



(c) HalfCheetah-v2, DDPG with PSN

Figure 10: Test

4 Conclusion

Above all, we can conclude that DQN and its variants is quite powerful to control games like Atari. Compared with DQN, the improvements like Double DQN, Dueling DQN and the combination of those improvements will further improve the performance and training efficiency. However, the DQN methods is not sample efficient, which means we need quite huge memory space to get good results. For A3C, I think we should avoid using it to train agents in environments with continuous action space, though it's more sample efficient and requires less computing resources. DDPG is a good algorithm to control agents in environments with continuous action space like HalfCheetah-v2. And its improvement DDPG with parameter space noise is powerful and can help to find global optimal solutions. But all the algorithms mentioned in the project can be affected by the differences among environments slightly or enormously. It means we need to adjust our algorithm in different environments and also carefully choose algorithms.

References

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI16, pp. 20942100. AAAI Press, 2016.

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. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML14, pp. I387I395. JMLR.org, 2014.

George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 1995–2003. JMLR.org, 2016.

A APPENDIX

You can pull the whole project from `https://github.com/DeanAlkene/CS489-ReinforcementLearning` which also includes codes and reports of 5 assignments. You can train the models by following command.

`python run.py --env_name` *env* `--method` *method*

All the models in my experiments are saved and can be tested using the command:

`python test.py`

However, you need modify the code of `test.py` to specify the model and environment you want to test. You can also watch the testing videos stored in `demo` folder.