Development of Warehouse Robot Arms for Grasping Objects

Project Milestone 2 for CS 5180 Renfrcmnt Lrning/Seq Decsn Mkg

Yichu Yang[*] and Wei Xu[*]

[*]Email: {yang.yich,xu.wei3}@northeastern.edu

1 Background

Nowadays, most of the industrial robots applied in various industries are doing repetitive tasks. Basically, moving or placing objects in a predetermined trajectories, which means we have identified every action that the robot is going to do. However, if the environment changes, or there are more complex tasks, these robots will be difficult to adapt, such as tying a node[4].

In the traditional control method, the robot arms are drove with precise mathematical model based on specific tasks. We need to consider about the control system designing, the error of each joint in inverse kinematics and the model accuracy when designing closed loop. When the task changes, the control system need to be changed accordingly to deal with the new task. In contrast, Reinforcement learning enables a robot autonomously discover an optimal behavior through trial-and-error interactions with its environment. Instead of explicitly detailing the solution to a problem, in learning method the designer of a control task provides feedback in terms of a scalar objective function that measures the one-step performance of the robot. Deep reinforcement learning (RL) has made great success in artificial intelligence games. In recent years, Deep RL has been introduced into the control of robot arm. Abbeel et al used the machine learning to robotics firstly, and he developed a robot capable of folding laundry, although very slowly.

Demand for robots is generally growing at a rapid clip, especially for the warehouse automation. Our finial goal is to develop a robot that can assist in grasping objects in the warehouse. The first step is to make the robot arm reach the target position accurately, and then grasp, which is also the goal for this project. We mainly used deep deterministic policy gradient (DDPG) and some other relative algorithms to train the robot arm.

2 Environment analysis

We will use the OpenAI Gym as the environment. It is a toolkit for developing and comparing reinforcement learning algorithms, which contains a robotics module. The robotics module need to connect to Mujoco. It is a physics engine aiming to facilitate research. It offers a unique combination of speed, accuracy and modeling power, yet it is a perfect simulator for this project.

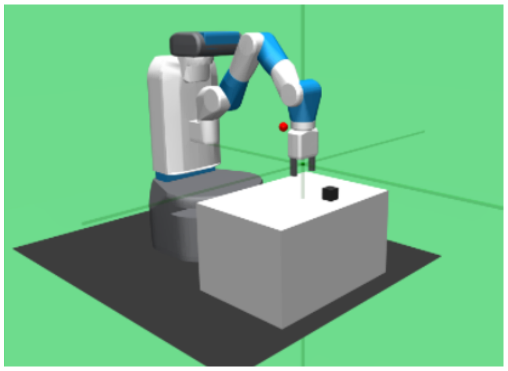

Figure 1: The *Fetch* environment in OpenAI Gym

In the Fetch environment, we can assign different mission to the Fetch robot, the available tasks are: Reach, Push, Pick and Place and Slide. Fetch robot is a 7-DOF robot equipped with a gripper. The agent obtains a **reward** of 0 if the object is at the target location (within a tolerance of 5 cm) and -1 otherwise. The **action space** for the agent is 4-dimensional, which indicates the 3-dimensional position in Cartesian space and the last for gripper action. The same action is applied to the robot in 20 subsequent simulator steps (with $\Delta \mathrm{t} = 0.002$ each) before returning control to the agent, thus the agent's action frequency is $\mathrm{f} = 25$ Hz. **Observations** include the Cartesian position of the gripper, its linear velocity, the position and linear velocity of the robot's gripper. If an object is present, we also include the object's Cartesian position and rotation using Euler angles, its linear and angular velocities, as well as its position and linear velocities relative to gripper.

3 Approach

3.1 Deep Deterministic Policy Gradient (DDPG)

The action space of this robotic environment is continuous, so discrete methods such as DQN will not perform well. So in this project, we explored a reinforcement learning algorithm named Deep Deterministic Policy Gradient(DDPG) that can handle continuous action space[3].

DDPG is devised by Google DeepMind. It is a solid algorithm for tackling the continuous action space problem. This algorithm is a policy-gradient actor-critic algorithm which is off-policy and model-free, and that uses some of the same methods from Deep Q-Networks (DQN).

The pseudo-code for DDPG algorithm is shown in Algo.1. For the aspect of deep, there are two important points. One is experience replay. Because of random sampling from the replay buffer, the data is more independent of each other and closer to i.i.d. Also, there are two networks, the policy network and the target network like DQN. Then the input and

Algorithm 1 Deep Deterministic Policy Gradient

```
 1: Randomly initialize critic network $Q\left(s, a|\theta^Q\right)$ and actor $\mu\left(s|\theta^\mu\right)$, with weights $\theta^Q$ and
    $\theta^\mu$
 2: Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$
 3: Initialize replay buffer $R$
 4: for $episode = 1 : M$ do
 5:    Initialize a random process $\mathcal{N}$ for action exploration
 6:    Receive initial observation state $s_1$
 7:    for $t = 1 : T$ do
 8:       Select action $a_t = \mu\left(s_t|\theta^\mu + \mathcal{N}_t\right)$ according to the current policy and exploration
          noise
 9:       Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
10:       Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
11:       Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
12:       Set
              $$y_i = r_i + \gamma Q^{'}\left(s_{i+1}, \mu^{'}\left(s_{i_1}|\theta^{\mu^{'}}\right)|\theta^{Q^{'}}\right)$$
13:       Update critic by minimizing the loss:
              $$L = \frac{1}{N}\sum_i \left[y_i - Q\left(s_i, a_i|\theta^Q\right)\right]^2$$
14:       Update the actor policy using the sampled policy gradient:
              $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q\left(s, a|\theta^Q\right)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu\left(s|\theta^\mu\right)|_{s_i}$$
15:       Update the target networks:
              $$\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\,\theta^{Q^{'}}$$
              $$\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\,\theta^{\mu^{'}}$$
16:    end for
17: end for
```

output are more stable to train the network. For the policy gradient, it is good at handling the continuous environment. Actually only one action is needed for each step. So the a deterministic way is applied to generate actions, making it easier to debug when conditions are poor.

The general workflow of DDPG is shown as Fig.2. It uses Agent-Critic concept, so there are two neural networks. In DDPG, the optimal action is generated deterministically by the Actor. The Actor follows the policy-based approach and learns how to act by estimating the optimal policy and maximizing reward through gradient descent. The Critic follows the value-based approach and learns how to estimate the value of different state-action pairs. By introduce a critic, the number of samples to collect for each policy update is reduced.

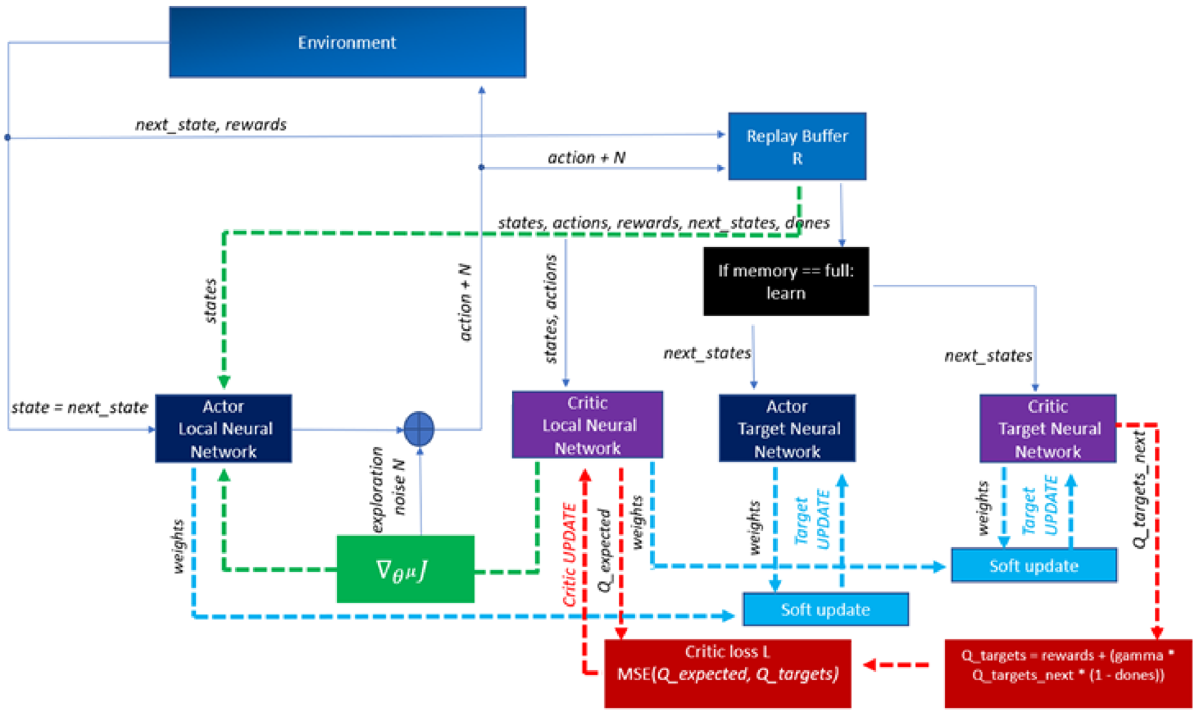

Figure 2: General overview of DDPG algorithm flow

The actor critic network structure we used in the implementation is shown in Fig.3. The critic and target critic network uses two 256-units hidden layers and is then activated using a ReLU function. Then the output is concatenated with action value and then put into a single 256-units layer and finally pass a ReLU function to get a state-action value estimation.

For the action network, we simply pass the observation through two 256-units hidden layer with both activated by ReLU and then a 4-units reshape layer with tanh activation to get 4 action output within -1 to 1. During the implementation, we found that the location of ReLU function matters, the structure shown in Fig.3 is found to be working correctly.

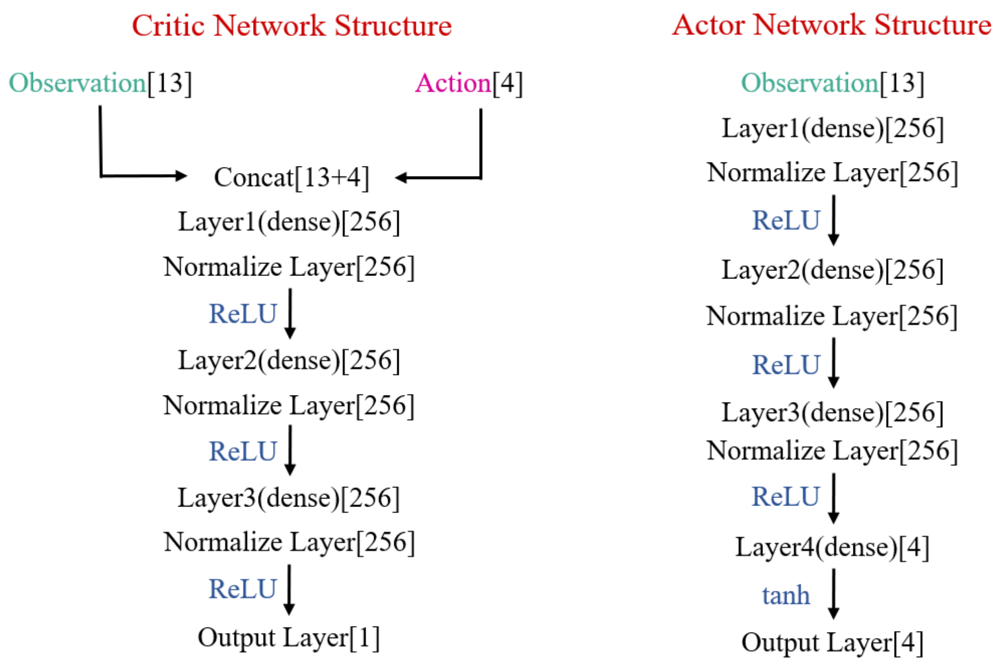

Figure 3: Actor and Critic Network Structure used in DDPG

3.2 Twin Delayed DDPG (TD3)

The goal of TD3 is to solve the overestimation bias problem in actor-critic methods due to the function approximation error[2]. While DDPG can achieve great performance sometimes, it is frequently brittle with respect to hyperparameters and other kinds of tuning. A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function. Twin Delayed DDPG (TD3) is an algorithm that addresses this issue by introducing three critical tricks:

- Clipped Double-Q Learning. TD3 learns two Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- "Delayed" Policy Updates. TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- 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.

Algorithm 2 Twin Delayed DDPG(TD3)

```
1: Input: initial policy parameters $\theta$, Q-function parameters $\phi_1$,$\phi_2$,empty replay buffer $\mathcal{D}$
2: Set target parameters equal to main parameters $\theta_{targ} \leftarrow \theta$, $\phi_{targ,1} \leftarrow \phi_1$,$\phi_{targ,2} \leftarrow \phi_2$
3: repeat
4:    Observe state s and select action $a = clip(\mu_\theta(s) + \epsilon, \alpha_{Low}, \alpha_{High})$, where $\epsilon \sim \mathcal{N}$
5:    Execute a in the environment
6:    Observe next state s', reward r and done signal d to indicate whether s' is terminal
7:    Store (s,a,r,s',d) in replay buffer $\mathcal{D}$
8:    If s' is termianl, reset environment state.
9:    if time to update then
10:       for $j$ in range(however many updates) do
11:          Randomly sample a batch of transitions, B=(s,a,r,s',d) from $\mathcal{D}$
12:          Compute target actions
                $$a'(s') = clip(\mu_{\theta_{targ}(s')} + clip(\epsilon, -c, c), \alpha_{Low}, \alpha_{High}), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$
13:          Compute targets
                $$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{targ,i}}(s', a'(s'))$$
14:          Update Q-functions by one step pf gradient descent using
                $$\nabla_\theta \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_{\phi_i}(s, a) - y(r, s', d))^2, \qquad \text{for i =1,2}$$
15:          if j mode policy_delay=0 then
16:             Update policy by one step of gradient ascent using
                   $$\nabla_{theta} \frac{1}{|B|} \sum_{s \in B} Q_{phi_1}(s, \mu_{theta}(s))$$
17:             Update target networks with
                   $$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1 - \rho)\phi_i \qquad \text{for i=1,2}$$
                   $$\theta_{targ} \leftarrow \rho\theta_{targ} + (1 - \rho)\theta$$
18:          end if
19:       end for
20:    end if
21: until convergence
```

The network structures of TD3 used in implementation is the same as those in DDPG, yet instead of only 4 networks(actor, critic, targert actor, target critic) we have 6 now(actor, critic1, critic2, targert actor, target critic1,target critic2).

3.3 Hindsight Experience Replay (HER)

While DDPG and TD3 performs greatly in many cases, they cannot handle the sparse reward very well. A sparse reward task is typically characterized by a meagre amount of states in the state space that return a feedback signal. A typical situation is a situation where an agent has to reach a goal and only receives a positive reward signal when he is close enough to the target. HER is how the networks are trained[1]. It is a new algorithm that can learn from failure. It can also learn successful policies from only sparse rewards. It does what humans do intuitively: even though we didn't get the specific goal, we pretend that we have reached the goal. By doing this, the algorithm gets some sort of learning signal since it has achieved some goal. In implementation it is done by adding such experience and fake goal pair to the replay buffer(as is marked in red in the algorithm) so that we can use them during training.

Algorithm 3 Hindsight Experience Replay

```
 1: Given: an off-policy RL algorithm $\mathbb{A}$, a strategy $\mathbb{S}$ for sampling goals for replay, and a
    reward function $r$: $\mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$
 2: Initialize $\mathbb{A}$
 3: Initialize replay buffer $R$
 4: for $episode = 1 : M$ do
 5:     Sample a goal $g$ and an initial state $s_0$
 6:     for $t = 0 : T - 1$ do
 7:         Sample an action $a_t$ using the behavioral policy from $\mathbb{A}$: $a_t \leftarrow \pi_b\left(s_t || g\right)$
 8:         Execute the action $a_t$ and observe a new state $s_{t+1}$
 9:     end for
10:     for $t = 0 : T - 1$ do
11:         $r_t := r\left(s_t, a_t, g\right)$
12:         Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in $R$
13:         Sample a set of additional goals for replay $G := \mathbb{S}$(current episode)
14:         for $g^{'} \in G$ do
15:             $r_t := r\left(s_t, a_t, g^{'}\right)$
16:             Store the transition $\left(s_t||g^{'}, a_t, r^{'}, s_{t+1}||g^{'}\right)$ in $R$
17:         end for
18:     end for
19:     for $t = 1 : N$ do
20:         Sample a minibatch $B$ from the replay buffer $R$
21:         Perform one step of optimization using $\mathbb{A}$ and minibatch $B$
22:     end for
23: end for
```

HER can be used together with any off-policy reinforcement learning algorithms. In our case, we use HER to enhance the performance of DDPG and TD3 algorithms on sparse-reward tasks.

4 Experiments and Results

In order to compare the advantages and disadvantages of various algorithms, we conducted the following experiments.

Table 1: Experiment design

algorithm	sparse reward		dense reward
	no HER	HER	
DDPG	DDPG	DDPG+HER	DDPG
TD3	TD3	TD3+HER	-

4.1 Reach task: Sparse reward vs Dense reward

The goal of Reach task is to learn a policy to move the robot end effector to the target position as close as possible. In order to have a more intuitive impression of the influence of sparse reward, we conducted a comparative study of dense reward and sparse reward. As can be seen from the Fig.4, the effect of sparse reward on training effect is huge. When the reward is dense, a satisfactory result can be obtained after about 150 episodes, while when the reward is sparse, it takes about 2000 episodes to get the similar result. The episodic reward is normalized so that we can get a more intuitive comparison. However the object

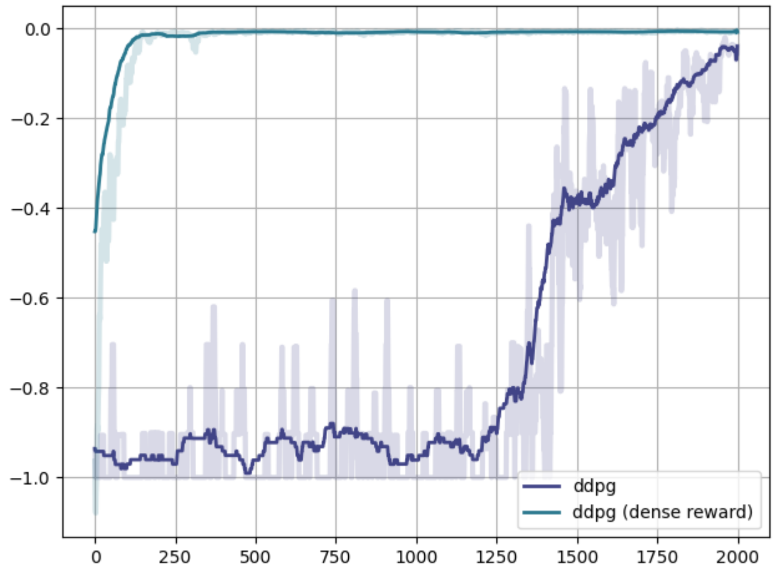

Figure 4: DDPG performance in dense reward and sparse reward environment

of our study is sparse reward environment since in reality it is hard to quantize the reward correctly, especially in multi-goal tasks. So in the rest of the project we test our algorithm in sparse reward environment only.

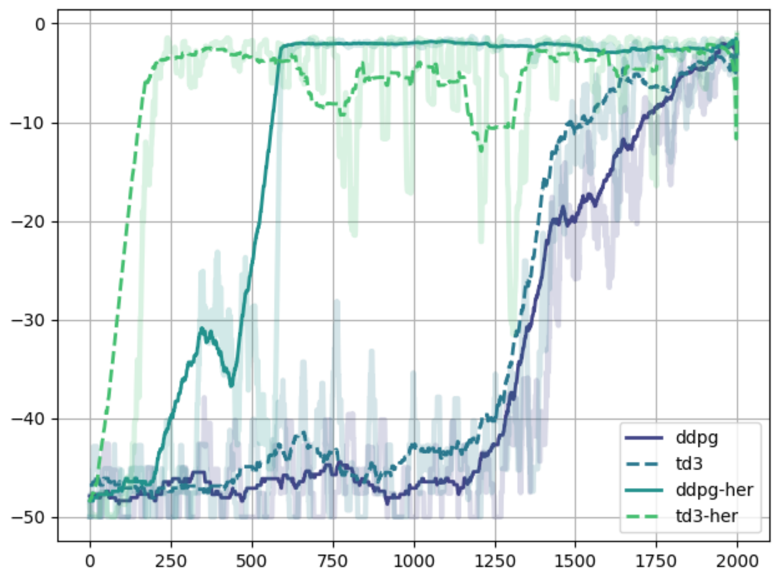

Figure 5: Comparison of DDPG and TD3 results

4.2 Reach task: DDPG, DDPG-HER, TD3 and TD3-HER

The result of TD3 implemented on gym fetch-reach task is shown in Fig.5 (bottom two lines). Both DDPG and TD3 agent successfully learned how to reach the target around 1500 episodes in the reach task while TD3 slightly outperforms DDPG.

After applying HER method to both of them, the results of TD3-HER and DDPG-HER are shown in Fig.5. The improvement is significant on both algorithm, the performance of TD3-HER is comparable with DDPG in dense reward environment.

The result of DDPG-HER is shown in Fig.5(top two lines), with the help of HER, DDPG agent is able to learn the correct action in 500 episodes and TD3 learns the policy within 200 episodes, the performance of TD3 is comparible to the DDPG performance in dense reward environment. So the combination of HER and off-policy RL algorithms can significantly improve the efficiency.

In the process of implementing the algorithms, we found that these algorithms are extremely sensitive to hyper-parameters. Due to the computational cost of the training process, tuning the parameters is a tedious and exploring job. So we take a look into the OpenAI's baseline code and we found a few tricks in improving the code and tuning efficiency in the following tasks:

- Environment sampling frequency can be different to learning frequency (in the original paper we learn immediately after interacting with the environment). In the baseline code, they run for multiple cycles per episode, in each cycle the agent samples the environment in n_rollouts times, and then updates for n_batchs time. So the total step they took should be:
  $n\_step = n\_episode * n\_cycles * n\_rollouts$
  That's why their agent seems to converge in significantly less epochs than our results, yet the total computational cost is close. In their case n_rollouts is twice as large

as n_batchs, so the agent is encouraged to gain more experience than learning. In additional, in baseline, the agent doesn't care about done or not and keeps sampling.

- Larger batch size, lower target updating ratio give more stable results but at the cost of computational time. The standard deviation of noise should not be large otherwise it is hard to converge.

4.3 Push Task: DDPG-HER vs TD3-HER

The goal of Push task is to learn a policy to use the robot arm to move the object to the target position as close as possible. So this is a more difficult problem than reach, it has two goals: reach the object and then move it. The results of DDPG-HER and TD3-HER

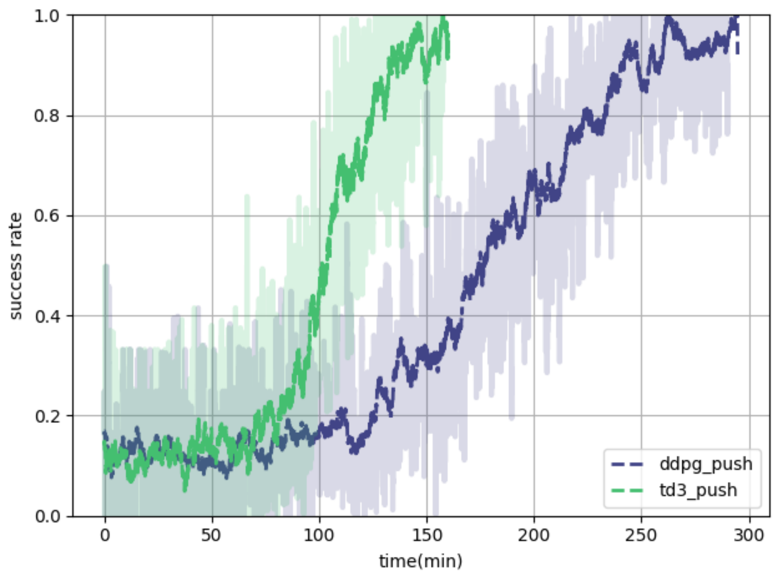

Figure 6: DDPG-HER and TD3-HER performance on Push task

are shown in Fig.6, with the help of HER, DDPG agent is able to learn the correct action in 300 minutes and TD3 learns the policy within 150 minutes.

Since we took the improvements in the baselines as mentioned above, the hyperparameters used in DDPG-HER and TD3-HER are different(mainly cycles and batch size) so the criteria of judgement we used here is different, instead of using episodes we use training time as the x-axis. Also, we use episodic reward as the y-axis before, now it is success rate we get from evaluation.

4.4 Pick and Place Task: DDPG-HER vs TD3-HER

The goal of Pick and Place task is to learn the policy to use the robot arm to pick up the object and then move it to the target position as close as possible. So this is a even more difficult problem than push, it has three goals: reach the object and pick it up then move it.

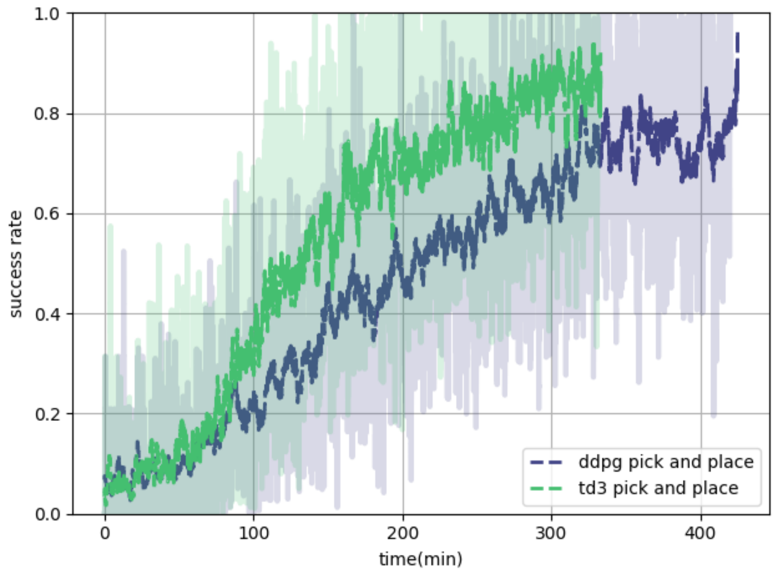

Figure 7: DDPG-HER and TD3-HER performance on Pick and Place task

The results of DDPG-HER and TD3-HER are shown in Fig.7, with the help of HER, DDPG agent is able to learn the correct action in more than 400 minutes and TD3 learns the policy within 300 minutes. The training time on pick and place task is significantly longer since it is more difficult which also make it harder to tune the hyperparameters.

5 Conclusion

In this project we explored the performance of DDPG and TD3 algorithms on continuous tasks with sparse reward. We also used HER (hindsight experience replay) method to augment the training efficiency of off-policy methods such as DDPG and TD3.

The performance of TD3 is always better than DDPG, with or without the help of HER. This is expected, but we also encountered a few unexpected problems when using TD3: TD3 is not as stable as DDPG with short delay time and low exploration rate, the success rate rises very quickly but drops back after some time. So we need to be much more careful when tuning the hyperparameters of TD3 since it has more parameters, with larger delay time and higher exploration rate, the algorithm tends to converge faster and more robust.

Although reinforcement learning method still needs a lot of tuning and training, it is an extraordinary approach especially in complicated control tasks where the environment cannot be modeled easily. To use traditional control methods in the task as above we need to calculate or do system identification to get dynamic parameters of the robot first then use optimization algorithms to get the trajectory. In our case the inverse kinematics of the robot is given which makes it easier for us to control the robot, the only thing left for us is to generate a trajectory for the PD controller. It seems like a simple problem, yet traditional control method will require a recomputation every time the target changes, yet using reinforcement learning ideas it is not necessary anymore. Furthermore, reinforcement learning method can not only take joint angles or end effector position as observation, it can also use camera input as observation directly, which is commonly known as end-to-end methods. In recent years, this is becoming a much more popular method in pick and place tasks because the shape of the object varies and traditional methods is more and more restricted. Yet using reinforcement learning doesn't mean we can completely discard traditional robotic control approach, sometimes the combination of physics and learning gives much better results and more robust convergence. Google's tossing bot is a good example, it can pick up and throw the object with high accuracy using a network based on physics controller[5].

So in conclusion, combination of reinforcement learning ideas and robotics is promising but there's still a long way to go.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, pages 5048–5058, 2017.

[2] Scott Fujimoto, Herke Van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[3] 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 preprint arXiv:1509.02971*, 2015.

[4] Jeremy Maitin-Shepard, Marco Cusumano-Towner, Jinna Lei, and Pieter Abbeel. Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding. In *2010 IEEE International Conference on Robotics and Automation*, pages 2308–2315. IEEE, 2010.

[5] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. 2019.