Multi-DDPG with Parameter Noise Exploration

Yuxin Han[1], Lianming Shi[2] and Chuqiao Song[3]

Abstract— **There are some deficiencies of traditional reinforcement learning algorithm, such as data inefficiency, computation inefficiency on single computer, unintelligent exploration problem, and not leveraging previous trained data. Recent algorithms, such as DDPG, A3C and TRPO solve one or some of these in some degree. The vanilla DDPG improves the exploration through Actor and Critic, and has a reply buffer to memorize samples including states, action and so on to leverage previous trained data. Adding noise based Ornstein Uhlenbeck process to action space is also an intelligent way to get a better exploration, which accelerates the convergence. However, it still can be get into the local minimum place and converge slowly even though implementing those mentioned above.**

In this project, the algorithm adds noise directly to the agents parameters to obtain a richer set of behavior and has a so called Multi-DDPG structure to choose best agent among agents in current episode as a distributor for next episode to reduce the chaos.

I. Introduction

Deep Reinforcement Learning has recently gained a lot of traction due to the significant amount of progress that has been made in the past few years. DQN makes people realize that deep learning methods could be used to solve high-dimensional problems, which means we do not need to know all things like MDP problem. Unfortunately, DQN can only deal with the discrete action space and most of environments are continuous. We cannot discretize the continuous action space too finely and discretization of the actions space throws away valuable information at the same time. Therefore, Google DeepMind produced a policy-gradient actor-critic algorithm called DDPG which is off-policy and model-free. By the way, it is worth to note that this algorithm derived from their prior work, Deterministic Policy Gradients [1].

DDPG is a combination of policy gradient and actor-critic algorithm. The speed of DDPG is related to the efficiency of exploration. It is proved that adding noise to DDPG helps algorithms efficiently explore the range of actions available to solve an environment. There are two main methods to add noise. One of the method is to add the noise to the action space directly to change the likelihoods associated with each action the agent might take from one moment to the next. Another one is to add adaptive noise to the parameters of the neural network policy rather than to the action space. The former one can only improve the efficiency of exploration, whereas the latter one can get out of the local minimum place as well, which is proved by HalfCheetah environment.

Even though we can avoid local minimum in HalfCheetah after using parameter noise, we cannot say that it can solve this problem for all environments. There is a possibility of falling into local minimum. In other words, there must be at least one time that agent does not fall into local minimum if we have enough test. Therefore, we are wondering if we can pick the best one from all agents. The idea is to run several agents at the same time and find the best one according to the total reward or loss. The replay buffer stores the experiences of all agents during training, which means we can get more available situations at one epoch. It is worth to note that those agents are running one by one in a epoch rather than in parallel as A3C [2]. The reason we do not implement it as D4PG [3] is that it is difficult to use GPU in parallel. Comparing with parallel CPU process, we prefer to use GPU to accelerate the speed to reduce the running time.

Our project considers a number of modifications to the DDPG algorithm. This algorithm has several properties that make it ideal for the enhancements we consider, which is at its core an off-policy actor-critic method. At first, the baseline of algorithm is applying noise directly to the neural network parameters. Parameter noise helps DDPG more efficiently explore the range of actions available to solve an environment. Training DDPG without parameter noise will frequently develop inefficient running behaviors, whereas policies trained with parameter noise often develop a high-scoring gallop. Furthermore, due to the fact that DDPG is capable of learning off-policy it is also possible to modify the way in which experience is gathered. In this work we utilize this fact to run many actors in parallel, all feeding into a single replay buffer. This allows us to seamlessly distribute the task of gathering experience.

II. Background

Traditional reinforcement learning algorithm always encountered problems of data inefficiency. Model-free value function approaches enable effective reuse of data and do not require full access to the state space or to a model of the environment. One recent work [4], closely related to the ideas followed in this paper, provides a proof of

[1]Yuxin Han is with graduate student of Electrical and Computer Engineering Department, University of California, San Diego, CA 92037, USA `Email: yuh379@ucsd.edu`

[2]Lianming Shi is with graduate student of Electrical and Computer Engineering Department, University of California, San Diego, CA 92037, USA `Email: l5shi@ucsd.edu`

[3]Chuqiao Song is with graduate student of Electrical and Computer Engineering Department, University of California, San Diego, CA 92037, USA `Email: chs140@ucsd.edu`

concept demonstration that value-based methods using neural network approximators can be used for robotic manipulation in the real world . This work applied a Q-learning approach [5] to a door opening task in which a robotic arm fitted with an unactuated hook needed to reach to a handle and pull a door to a given angle. This task was learned in approximately 2 hours across 2 robots pooling their experience into a shared replay buffer.

This work thus made use of a complementary solution to the need for large amounts of interaction data: the use of experimental rigs that allow large scale data collection, including the use of several robots from which experience are gathered in parallel. This can be combined with single machine or distributed training depending on whether the bottleneck is primarily one of data collection or also one of network training.

A. DPG and DDPG

DPG is a policy gradient algorithm for continuous action spaces that improves the deterministic policy function via back propagation of the action-value gradient from a learned approximation to the Q -function. Specifically, DPG maintains a parametric approximation $Q(x_t,u_t;\phi)$ to be the action value function $Q^{\pi}(x_t,u_t)$ associated π and ϕ is chosen to minimize

$$E_{(x_t,u_t,x_{t+1})\sim\bar{\rho}}[(Q(x_t,u_t;\phi)-y_t)^2] \tag{1}$$

where $y_t=r(x_t,u_t)+\gamma Q(x_{t+1},\pi(x_{t+1})$. $\bar{\rho}$ is usually close to the marginal transition distribution induced by π but often not identical. For instance, during learning u_t may be chosen to be noisy version of $\pi(x_t,\theta)$, e.g. $u_t=\pi(x_t;\theta)+\varepsilon$ where $\varepsilon\sim N(0,\sigma^2)$ and $\bar{\rho}$ is then the transition distribution induced by this noisy policy. The policy parameters are then updated according to

$$\Delta\theta \propto E_{(x,u)\sim\rho}[\frac{\partial}{\partial u}Q(x,u;\phi)\frac{\partial}{\partial\theta}\pi(x;\theta)] \tag{2}$$

DDPG is an improvement of the original DPG algorithm adding experience replay and target networks: Experience is collected into a buffer and updates to θ and ϕ are computed using mini-batch updates with random samples from this buffer. Furthermore, a second set of "target- networks" is maintained with parameters θ' and ϕ'. Both measures significantly improve the stability of DDPG. The structure of DDPG is shown in Figure 1

B. Parameter Noise

Parameter noise adds adaptive noise to the parameters of the neural network policy, rather than to its action space. The comparison between action space noise and parameter space noise is shown in Figure 2 Traditional Reinforcement Learning uses action space noise to change the likelihoods associated with each action the agent might take from one moment to the next. Adding noise to the action space leads to more unpredictable exploration which isnt correlated to anything unique to the agents parameters. Parameter space

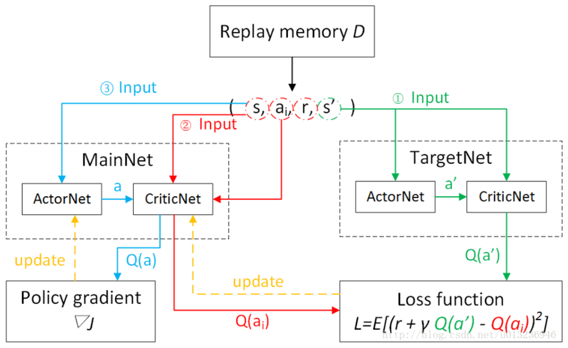


Fig. 1: The structure of DDPG

noise injects randomness directly into the parameters of the agent, altering the types of decisions it makes such that they always fully depend on what the agent currently senses. Adding noise in a deliberate manner to the parameters of the policy makes an agents exploration consistent across different time steps.

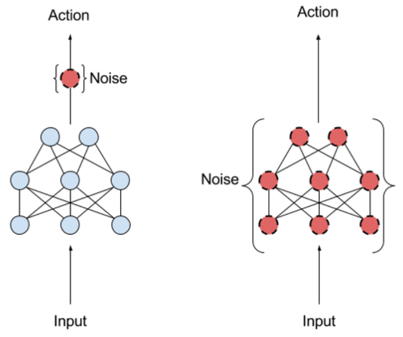


Fig. 2: Action space noise compared to parameter space noise

III. METHODS

The procedure of our project is straightforward, and it just combines parameter noise exploration with N DDPG structures along with some modification. At the beginning, we use DDPG structure to create a distributor agent and N running agents. Then for each episode, we add parameter noise onto the actor neural net of the distributor agent, which is for policy π_θ, by using $\tilde{\theta}=\theta+N(0,\sigma^2 I)$. Also, the σ value has to be initialized at beginning and scaled along with each episode. Therefore, we can use the action generated from this perturbed policy $\pi_{\tilde{\theta}}$ to explore the environment. Next, the weights of distributor agents (actor, critic, target actor, and target critic) are copy to N running agents, and the initial state s_0 is generated for each running agent.

For each step in one episode, we first sample an action a_t from the perturbed actor net of each running agent with Ornstein-Uhlenbeck noise. Then we execute the action, respect to each running agent, to explore the environment to obtain r_t and s_{t+1} receptively. Also, we store N generated (s_t,a_t,r_t,s_{t+1}) into the replay buffer,and set $s_t=s_{t+1}$

for next iteration. By generating these independent experiences(because each agent is exploring their own environment individually) into the replay buffer, we can accelerate replay buffer refreshing and make data more efficient. To update our N running agents, we first sample N mini-batches from the replay buffer, and we use these mini-bathes to update each running agent respectively by using following two equations, which are policy gradient for actor net, and loss calculation for critic net.

$$\nabla_{\tilde{\theta}^{\pi}} \approx = \frac{1}{K}\sum_{i}^{K} \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\pi_{\tilde{\theta}}(s_i)} \nabla_{\tilde{\theta}^{\pi}} \pi(s_i|\tilde{\theta}^{\pi}) \quad (3)$$

$$L = \frac{1}{K} = \sum_{i}^{K} (y_i - Q(s_i,a_i|\theta^Q))^2 \quad (4)$$

In general, we have to loop above step procedure until it meets the break condition. At the end of each step, the running agent with largest total reward would be selected as next distributor agent for next episode. At here, we also need a distance measure between the beginning unperturbed policy and perturbed policy of the selected running agent. Therefore, we collect all generated and initial states in this episode using $d(\pi, \pi_{\tilde{\theta}}) = \sqrt{\frac{1}{G} = \sum_i^G E_s[(\pi_\theta(s_i) - \pi_{\tilde{\theta}}(s_i))^2]}$ where $G = \sum_i^N K_i$. After calculating the distance, we can calculate scaled σ_{j+1} for next episode.

$$\sigma_{j+1} = \begin{cases} \alpha\sigma_j, & \text{if} d(\pi, \pi_{\tilde{\theta}}) < \delta \\ \frac{1}{\alpha}\sigma_j, & \text{otherwise} \end{cases} \quad (5)$$

For more detail, please check the pseudo-code below at algorithm 1. The next result section, we use $N = 3$ for number of running agents, initial $\sigma = 0.1$, $\delta = 0.2$, and $\alpha = 1.01$ to implement our Multi-DDPG with parameter noise exploration algorithm.

IV. RESULTS

In this section we compare the performance of the Multi-DDPG algorithm with original DDPG and DDPG with parameter noise across a variety of mujoco continuous control tasks, such as pendulum, inverted pendulum, half-cheetah, and double inverted pendulum environments, which are shown in Figure 3.

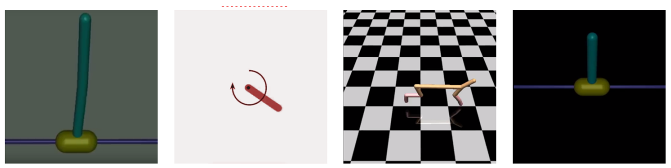
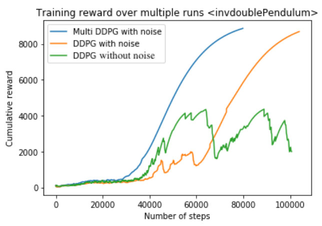


Fig. 3: InvertedDoublePendulum, Pendulum, HalfCheetah and InvertedPendulum

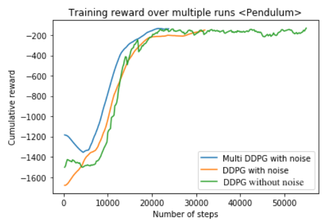
As shown in Figure 4, Multi-DDPG with parameter noise scales robust and outperform other methods with increasing action dimensionality and less iteration to converge. What's

Algorithm 1 Multi-DDPG with Parameter Noise Exploration

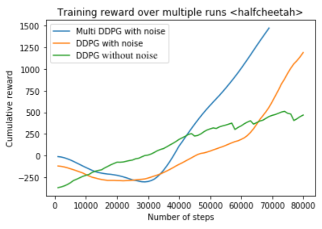
```
 1: create a distributor agent and N agents
 2: for Episode = 0 to M do
 3:     add parameter noise onto distributor agent
 4:     copy weights of the distributor agent to N agents
 5:     init $s_0$ for each agents
 6:     while s $\neq$ done for each agent And num-steps < threshold do    ▷ stop following procedure for the agent arriving at done condition
 7:         select $a_t = \pi(s_t|\tilde{\theta}) + OU_{noise}$ for each agent
 8:         execute $a_t$ and observe $r_t$ and $s_{t+1}$ for each agents
 9:         store N $(s_t, a_t, r_t, s_{t+1})$ into replay buffer
10:         set $s_t = s_{t+1}$ for each agent
11:         sample N mini-batches from replay buffer
12:         update N agents with generated batches receptively
13:     select distributor agent among N agents ▷ based on total reward of each agent
14: return distributor agent
```



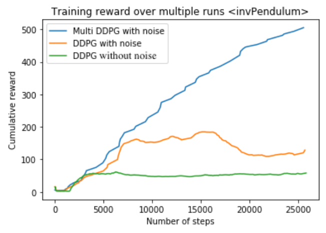
(a) InvertedDoublePendulum



(b) Pendulum



(c) InvertedDoublePendulum

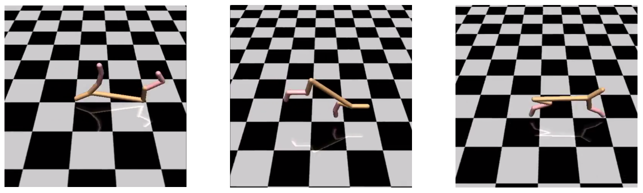


(d) Pendulum

Fig. 4: Rewards with steps

more, the decline of reward of Multi-DDPG with parameter noise in (c) at very beginning shows that Multi-DDPG with parameter noise can seek more available situations because of the replay buffer. In other words, the more situations learned at beginning, the better performance later. Here we take a close look at the performance of Multi-DDPG with parameter noise on Half-Cheetah environment in Figure 5.

We find that, in this environment, DDPG without parameter noise always quickly converges to a local optimum (in which the cheetah learns to flip on its back and then wiggles its way forward). After applying parameter noise to DDPG, the cheetah behaves similarly initially but still explores other options and quickly learns to break out of this local optimal behavior and perform forward running but in a shaky way.



(a) DDPG without parameter noise (b) DDPG with parameter noise (c) Multi-DDPG with parameter noise

Fig. 5: Performance of HalfCheetah with different methods

Multi-DDPG with parameter noise, in this case, achieves significantly higher returns than all other configurations. That makes sense because this algorithm allows data collection and network training to be distributed. Thus, Multi-DDPG helps to guide the learning process towards good solutions and reduce the pressure on exploration strategies and speed up learning. Hence, that is why our algorithm always develop a high-scoring gallop among the three methods.

V. DISCUSSION & CONCLUSIONS

We propose our baseline algorithm, parameter space noise, as a conceptually simple yet effective replacement for traditional action space noise like ε-greedy additive Gaussian noise. This work shows that parameter perturbations can successfully be combined with contemporary off-policy deep Reinforcement Learning algorithms DDPG and often results in improved performance compared to action noise. Also, by implementing Multi-DDPG structure, we can stabilize the variance of the vanilla DDPG parameter noise method. Experimental results further demonstrate that using parameter noise with Multi-DDPG allows solving environments with very sparse rewards quickly, in which action noise is unlikely to succeed. Our results indicate that parameter space noise is a viable and interesting alternative to action space noise, which is still the de facto standard in most reinforcement learning applications.

.

REFERENCES

[1] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[3] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap, "Distributed distributional deterministic policy gradients," *CoRR*, vol. abs/1804.08617, 2018. [Online]. Available: http://arxiv.org/abs/1804.08617

[4] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 3389–3396.

[5] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International Conference on Machine Learning*, 2016, pp. 2829–2838.