Reacher project report

Part 1 Adapted DDPG algorithm

Since the Reacher environment has 2 different versions, one with single agent, the other with multiple agents, I referenced the Udacity benchmark implementation and the gcolmen's GitHub and created a one-fit all code structure for both environments. The pseudocode for the adapted DDPG algorithm is as following:

Algorithm: Adapted Deep Deterministic Policy Gradient (see openai DDPG document)

0: Hyperparameters: Train_every (for every Train_every steps, train the agent); N_learn_updates (update N_learn_updates times for each learning process)

1: Input: initial policy network (Actor) with parameters θ, Q-function network (Critic) with parameters ϕ, empty replay buffer

D, agent size AS

2: Set the target networks' parameters equal to main parameters $\theta_{target} \leftarrow \theta, \phi_{target} \leftarrow \phi$

3: episode_count = 0

4: Repeat

5: Observe state s and select action a = $\text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where ϵ is the noise generated by a noise function of some kind, which will be discussed later.

$$\dim(a) = AS \times action\_size$$

6: Execute *a* in the environment

7: Observe next state s', reward r, and done signal d to indicate whether s' is terminal

8. Calculate average reward for all agents $r_{avg} = \frac{1}{AS}(\sum_i r_i)$

9: Store (s,a, r_{avg},s',d) in replay buffer D

10. episode_count += 1

11: If s' is terminal, reset environment state.

12: **If** episode_count % Train_every_steps == 0 **then**

13: **for** N_learn_updates **do**

14: Randomly sample a batch of transitions, B = {(s,a,r_{avg},s',d)} from D

15: Compute targets

$$y(r_{avg}, s', d) = r + \gamma(1 - d)Q_{\phi_{targ}(s', \mu_{\theta_{target}}(s'))}$$

Update Q-function by one step of gradient descent using

$$\nabla_{\phi} \frac{1}{|B|} \sum_{(s,a,r_{avg},s',d)\in B} \left(Q_{\phi}(s,a) - y(r,s',d)\right)^2$$

Update policy by one step of gradient ascent using

$$\nabla_{\theta} \frac{1}{|B|} \sum_{s\in B} Q_{\phi}(s, \mu_{\theta}(s))$$

Update target networks with

$$\phi_{\text{target}} \leftarrow \tau\phi_{target} + (1-\tau)\phi$$

$$\theta_{\text{target}} \leftarrow \tau\theta_{target} + (1-\tau)\theta$$

end for

end if

until environment solved

Part 2 Basic Neural Network design and Hyperparameters selection

Part 2.1: Neural Network architecture

Actor network:

Layer	Layer type	Input size	Output size	Activation function	Parameter Initialization
1	Fully connected	State size	fc1_units = 128	Relu	Uniform distribution $[-\frac{1}{\sqrt{state\ size}}, \frac{1}{\sqrt{state\ size}}]$
2	Batch normalize	128	128	/	/
3	Fully connected	128	fc2_units = 256	Relu	Uniform distribution $[-\frac{1}{\sqrt{128}}, \frac{1}{\sqrt{128}}]$
4	Fully connected	256	Action size	tanh	Uniformly sampled within $[-3\times10^{-3}, 3\times10^{-3}]$

Critic network:

Layer	Layer type	Input size	Output size	Activation function	Parameter Initialization
1	Fully connected	State size	fc1_units =128	Relu	Uniform distribution $[-\frac{1}{\sqrt{state\ size}}, \frac{1}{\sqrt{state\ size}}]$
2	Batch normalize	128	128	/	/

3	Fully connected	128+action size	fc2_units 256	Relu	Uniform distribution $[-\frac{1}{\sqrt{128+\text{action size}}}, \frac{1}{\sqrt{128+\text{action size}}}]$
4	Fully connected	256	Action size	/	Uniformly sampled within $[-3 \times 10^{-3}, 3 \times 10^{-3}]$

The hyperparameters fc1_units, fc2_units are chosen based on the discussion (require udacity account to have access).

The parameter initialization method is based on section 7 of the DDPG Paper.

Part 2.2 General Hyperparameters

The hyperparameters in the table below are general hyperparameters that are shared by both my Ounoise implementation and Gaussian noise implementation to solve the environment.

Hyperparameter	Value	Usage	Reason for choosing
Buffer size	10^6	Size of the replay buffer	Value suggested by section 7, DDPG Paper
Batch size	128	Size of each sample from the replay buffer	Default value of the DDPG implementation of Udacity drlnd
Gamma	0.95	Discount factor	Referenced gcolmen's GitHub
Tau	10^{-3}	Coefficient for soft target updates	Value suggested by section 7, DDPG Paper
LR_ACTOR	10^{-4}	Learning rate for the actor network	Referenced gcolmen's GitHub
LR_CRITIC	10^{-3}	Learning rate for the critic network	Referenced gcolmen's GitHub
WEIGHT_DECAY	0	Weight decay for critic network	Suggested by this discussion (require Udacity account to access)
TRAIN_EVERY	20	The agent(s) will learn every TRAIN_EVERY timesteps	Suggested by Udacity benchmark implementation
N_LEARN_UPDATES	10	The agent(s) will update N_LEARN_UPDATES each learning process	Suggested by Udacity benchmark implementation

Part 2.3: Noise implementation and Hyperparameters

Ounoise implementation

The Ounoise implementation generates the noise factor ϵ for action selection (line 5 of the pseudocode) by the

Hyperparameter	Value	Usage	Reason for choosing
Theta	0.15	Parameter for Ounoise	Referenced gcolmen's GitHub
Sigma	0.08	Parameter for Ounoise	Referenced gcolmen's GitHub

Using this set of hyperparameters, the 20-agent environment is solved by 108 episodes for the environment. The single agent environment is solved by 592 episodes.

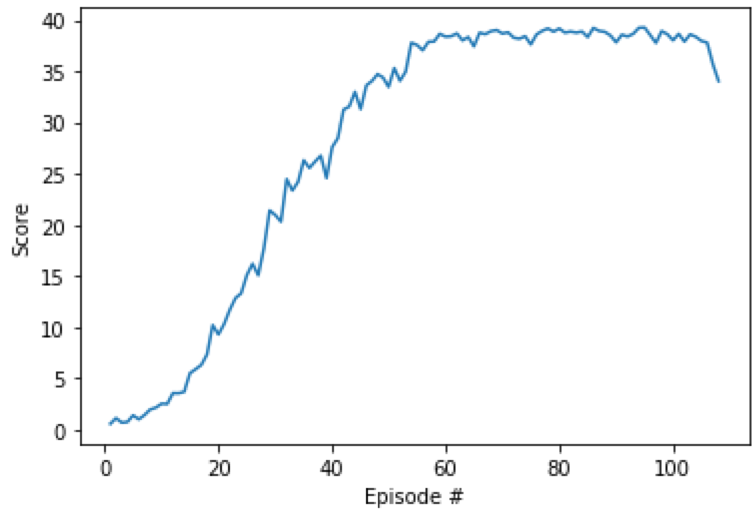


Figure 2.3-1: the OUNoise implementation solves the 20-agent environment in 108 episodes

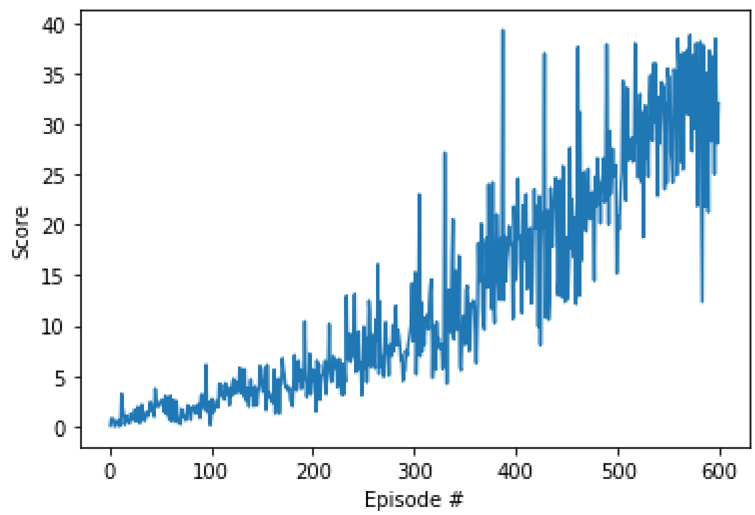


Figure 2.3-2: the OUNoise implementation solves the single agent environment in 592 episodes

Gaussian noise implementation

As stated in the "Exploration and Exploitation" section of the OPENAI DDPG document, *uncorrelated, mean-zero Gaussian noise works perfectly well* for DDPG. Based on this idea, I implemented a solution in which the noise is directly generated by gaussian random number, the only parameter involved here is sigma, square root of the variance.

Hyperparameter	Value	Usage	Reason for choosing
sigma	0.95	square root of the variance	By my own experiments

Using this set of hyperparameters, the single agent environment is solved by 191 episodes.

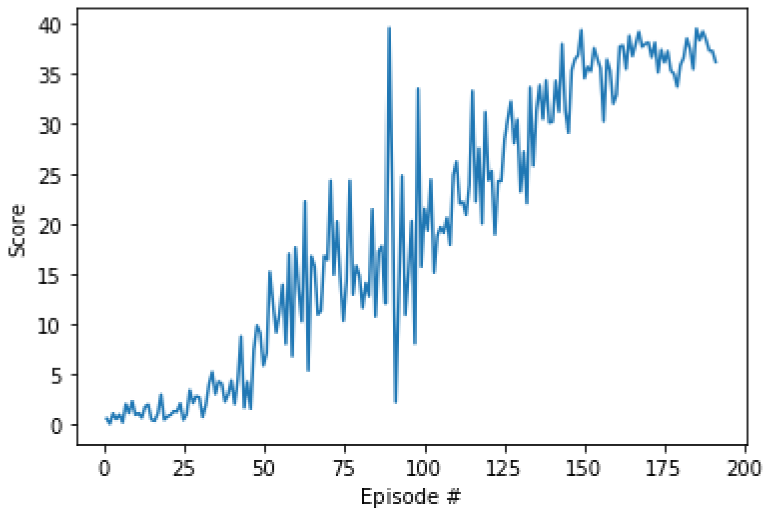


Figure 2.3-3: the Gaussian implementation solves the single agent environment in 192 episodes.

Part 3 Comparison between different implementation

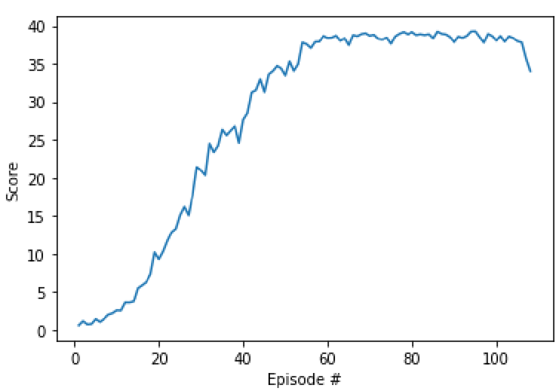
All the hyperparameters I listed in part 2 are "optimal" parameters that solve the environment as fast as possible. In this section, I'd like to compare implementations with different environment, design and hyperparameters, and discuss a few points that may be essential for solving the environment using DDPG.

Index	Implementation	Environment	Hyperparameters	Episodes for solving	Real time for solving
1	Ounoise	20 agents	Theta = 0.15; Sigma = 0.08;	108	Roughly 4 hours
2	Ounoise	Single agents	Theta = 0.15; Sigma = 0.08;	592	Roughly 1.5 hours
3	Ounoise	Single agents	Theta = 0.5; Sigma = 1;	Not solved within 600 episodes	The score is steady around +22 after 400 rounds

4	Ounoise	20 agents / single agents	Theta = 0.15; Sigma = 0.08; But white noise for Ounoise process generated by "random.random()"	Average score never > 1	The white noise generated is always positive, the agent never works
5	Gaussian	Single agents	Sigma = 0.9	303	Roughly 1 hour
6	Gaussian	Single agents	Sigma = 0.925	245	Roughly 45 minutes
7	Gaussian	Single agents	Sigma = 0.95	191	Roughly 35 minutes
8	Gaussian	Single agents	Sigma = 0.975	265	Roughly 50 minutes
9	Gaussian	Single agents	Sigma = 0.95 With priotized experience replay $\alpha = 0.3$	264	Roughly 50 minutes
10	Gaussian	Single agents	Sigma = 0.95 With priotized experience replay $\alpha = 0.1$	197	Roughly 35 minutes

Based on the experiments, I found several points that may be essential for solving the task:

1. The DDPG behavior is quite sensitive to the noise we add, a small change in parameters for noise generation may cause the agent to solve the task much slower or even cannot start learning. The noise must be centered at 0, with no bias. (There exists an error in line 150 of the Udacity's ddpg agent implementation. The white noise generated in the OUNoise class should not use random.random(), which is a value between 0 and 1)
2. The Gaussian noise implementation works better than the Ounoise implementation, since the agent converges much quicker.
3. The 20-agents environment produces a more stable result than the single agent environment.



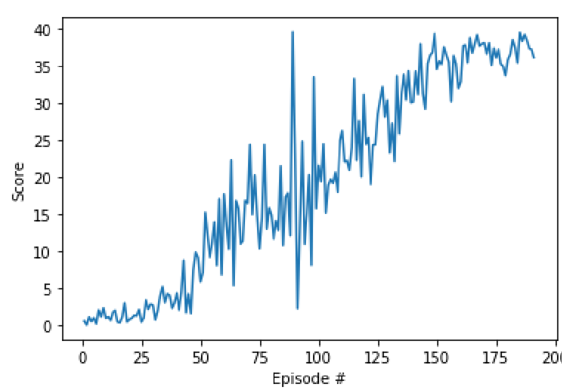


Figure 3-1: left is the training score plot of the multi-agent environment, right is the training score plot of the single agent environment. Clearly, the multiple-agent score plot is much smoother than the single one.

Part 4 Future improvement discussion

I paid much emphasis on tuning the design and hyperparameters for the noise generation for this project, and didn't pay much attention to the design of the neural network and other hyperparameters. Maybe tuning those parameters will give better results