For project, most of DDPG codes have been re-used from project 2. In addition, a multi-agent class is newly implemented.

Background

The main code is inspired by :
https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum

Algorithms

In this project, a modified DDPG is in place to train the agent. The main idea of DDPG is shown below. Basically, there are two actor networks (local and target) to learn the policy, using (state, action). There are another two critic networks (local and target) to learn the Q-value function, using the action generated from actor network. Then use a soft-update to update target model for both actor and critic.

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

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

      Update Q-function by one step of gradient descent using

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

      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

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1-\rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$

    end for
  end if
until convergence

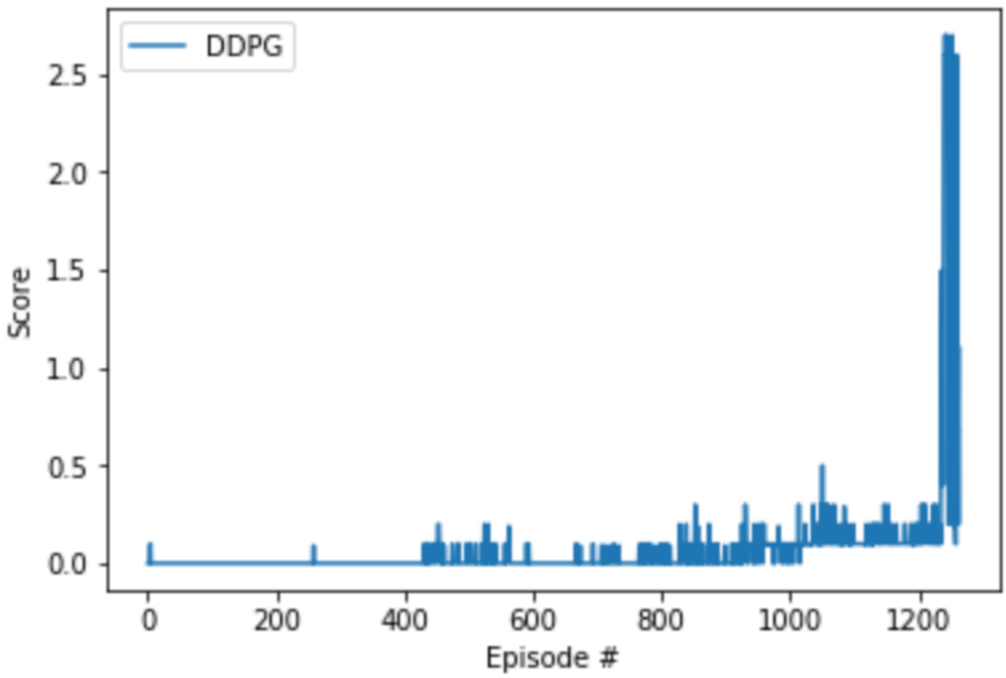
Lessons learnt while tuning the hyper-parameters

1. Use normalization for neural network input(this is also suggested from original paper). This will help converge faster
2. Ornstein-Uhlenbeck process noise generation, using a smaller sigma helps to converge (use 0.05 here).
3. Using update interval from “benchmark implementation”: update the networks 3 times after every 2 timesteps.
4. Use clipping policy for critic model to avoid cliff jumping.
5. Add a noise decay for generating the action.

(Hyper)parameters

Actor model	Neural network
	First linear layer: input 24 neurons, output 256 neurons. Activation function: Relu
	Second linear layer: input 256 neurons, output 128 neurons. Activation function: Relu
	Third linear layer: input 128 neurons, output 2 neurons. Activation function: tanh
	Note. A normalization is done before Relu from first to second layer.
Critic Model	Neural network
	First linear layer: input 24 neurons, output 256 neurons. Activation function: Relu
	Second linear layer: input 258 neurons, output 126 neurons. Activation function: Relu
	Third linear layer: input 126 neurons, output 1 neurons. Activation function: tanh
	Note. A normalization is done before Relu from first to second layer.
Replay buffer size	1000000.0
Batch size	256
Discount factor, Gamma	0.99
Actor model learning rate	0.0001
Critic model learning rate	0.0001
Weight decay	0
Soft update TAU	0.001
Learning internal	2
Learning num	3
Noise epsilon	1
Noise epsilon decay	0.999

With current strategy, after 1264 episode, the score is steady to 0.51(see the plot below).



Next step

1. The neural network architecture could be fine-tuned.
2. Tune other hyper-parameters (eg. learning step) to achieve the goal faster.
3. Try out other methods, eg. Monte Carlo Tree Search, etc.
4. The max score of each episode starts to decrease after 1255 episode, do more research and see how to stabilize this.