Machine Learning Engineer Nanodegree

Capstone Proposal

Ronin Ho

June 9th, 2019

Proposal

Domain Background

In this project, I am going to make use of the reinforcement learning to implement an agent which can control a lander to land on a designated area. The environment used is OpenAI Gym LunarLanderContinuous-v2[1]. Undoubtedly, Reinforcement learning is the one of the interesting topic in machine learning. Without explicitly instructing an agent how to act in the environment, instead, allowing it to check with the environment, make a decision, take an action and check how the environment response and learn from the experience. This is similar to how human is growing by learning from the past experience.

On May 4, I watched a live SpaceX mission on youtube[2], two rockets were fired to space to deliver a satellite and then successfully returned to a landing pad. It was amazing that the rockets can return to the ground and land automatically and without human instructions. With this technique, the rocket can be reused at next time. In addition, this can save the cost and save the environment from dropping any disposal into the sea.

Problem Statement

A lander and a landing pad are provided in the environment. A landing pad is located at coordinates (0,0) while the lander will be located at the top of the environment. Agent has to learn flying and controlling the lander to land on the landing pad. Agent should observe the current coordinates of the lander and then take the actions which are the powers of main engine and side engine.

The power of both engine is a continuous real values from -1 to 1. The episode ends when the lander crashes or comes to rest.

The reward will be given in below conditions.

Condition	Rewards
Lander crashes	-100
Landers fires main engine	-0.3 per frame

Lander comes to rest	+100
Lander’s leg gound contact (per leg)	+10
Landers land of the pad	+200

Datasets and Inputs

Data and inputs are not required to prepare in this reinforcement learning project. The input stage vectors are provided by the LunarLanderContinuous-v2 environment, while the action values are determined by the agent.

Solution Statement

Deep Deterministic Policy Gradient (DDPG) Algorithm will be used to implement the agent to work in this environment with continuous action space . DDPG makes use of deep neural networks to compute the state-action values (Q-values) and learn a policy respectively.

When the action space is discrete, the number of available actions is limited. In most of the cases, we can adopt a value-based learning policy by selecting an action which has the maximum Q-value for a given state. We can compute the Q-value of each action separately and then simply compare them. However, when the action space is continuous, the number of available actions seems to infinite. Even we can break them into a lot of very small intervals, it is still expensive to compute all of the possible Q-values. It will be unacceptable for an agent to spend so much resources to take an action for a single stage only.

In contrast, DDPG does not directly rely on the Q-value to learn a policy. It uses Policy Gradients, a policy-based method, to find the optimal policy. This skips the process of calculating the Q value of all possible actions.

DDPG contains deep neural networks which are Q-network and Policy network. Q-network is used to predict the optimal Q-value of a state and action pair. Policy network is used to predict the best action for a state. When learning an experience, DDPG predicts
1. Q-value before taking aaction for a visited state
2. Q-value after taking the action for a visited state by using the reward received and the Q-value of next state

The 2nd Q-value acts as a target value which Q-network should be predicted as closer as possible. By calculating 2 Q-values difference, Q-network is able to know how much error in the prediction. Alos, a gradient of the Q-values with respect to actions can be found to indicate how to minimize the error. This gradient can help to evaluate the quality of the action chosen by the policy for a state. Then, the Policy network can follow the gradient and update its network parameter and make better action.

Hence, DDPG works in continuous action-spaces environment.

Benchmark Model

OpenAI Gym environment LunarLanderContinuous-v2 defines "solving" as getting average reward of 200 over 100 consecutive trials. The same benchmark will be used in this project in order to consider my agent can solve the problem.

Evaluation Metrics

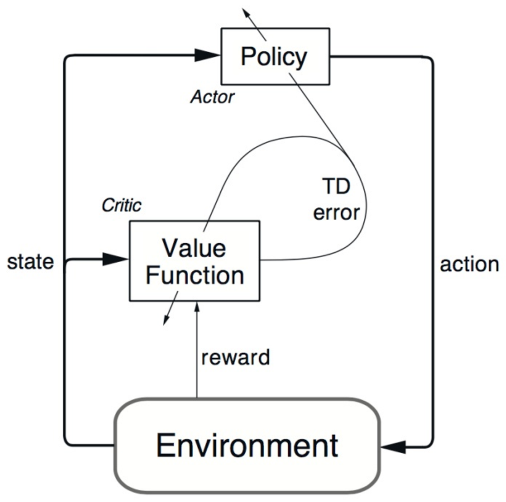
OpenAI Gym leaderboard[3] for environment LunarLanderContinuous-v2 will be used to compare and check my agent performance.

The current leaderboard is

Ranking	Episodes
1st	100
2nd	1500
3rd	5000
4th	5300

Project Design

1. Actor-Critic architecture will be used to implement the DDPG alogrithum.



Actor-critic Architecture[4]

Actor consists of a deep neural network called policy network and it is used to output the optimal actions based on the input state. While the Policy network learns from the action gradient from Critic and then updates its own network parameters.

Critic also consists of a deep neural network called Q network and it acts as value function and is used to predict the optimal Q value based on the state and the action. It also provide a method for Actor to access the gradient of Q value with respect to action.

2. Ornstein-Uhlenbeck Noise
   DDPG trains a deterministic policy which always gives the optimal policy as of what it visited. In the beginning of training, it may able to find a good action, but may not be the optimal. Since it does not try a wide enough variety of actions to find the optimal actions. To make DDPG policies explore more state action combinations, an Ornstein-Uhlenbeck Noise (OU-Noise) will be added to the actions at the training time. It essentially generates random samples from a Gaussian distribution, but each sample affects the next one such that two consecutive samples are more likely to be closer together than further apart. This can give us some similar noise values during adjacent states.

3. Target Network
   A target network technique will be used in the DDPG. As mentioned in the solution statement, when DDPG learns from experience, it predicts the
   1. Q value before the current action taken
   2. Q value (target Q value) after the current action taken.

   To calculate the target Q value, it involves predicting the action taken on next state by the policy network and the corresponding Q value.

$$Q_{current} = Q(s, P(s))$$
$$Q_{target} = r + d*Q(s', P(s'))$$

   where
   Q = Q network
   P = Policy network, P(s) is the action to be taken
   r = reward
   d = discount factor
   s' = next state

   In the learning process, we are trying to minimize the loss/difference between these 2 values and train $Q_{current}$ be more like the Q_{target}. As the target Q value is using the same network parameters as used in the training, when updating the Q network and Policy network parameters, the target values is also being updated. It likes just chasing a moving target. This will make the training unstable.

To tackle this problem, 1 more network will be added to Actor and Critic respectively, namely target networks. Target networks are similar to original Q network and Policy network, but they are using a separate set of network parameters from Q network and Policy network.

$$Q_{current} = Q(s, P(s))$$
$$Q_{target} = r + d*Q'(s', P'(s'))$$

where
Q = Q network
Q' = Q target network
P = Policy network, P(s) is the action to be taken
P' = Policy target network
r = reward
d = discount factor
s' = next state

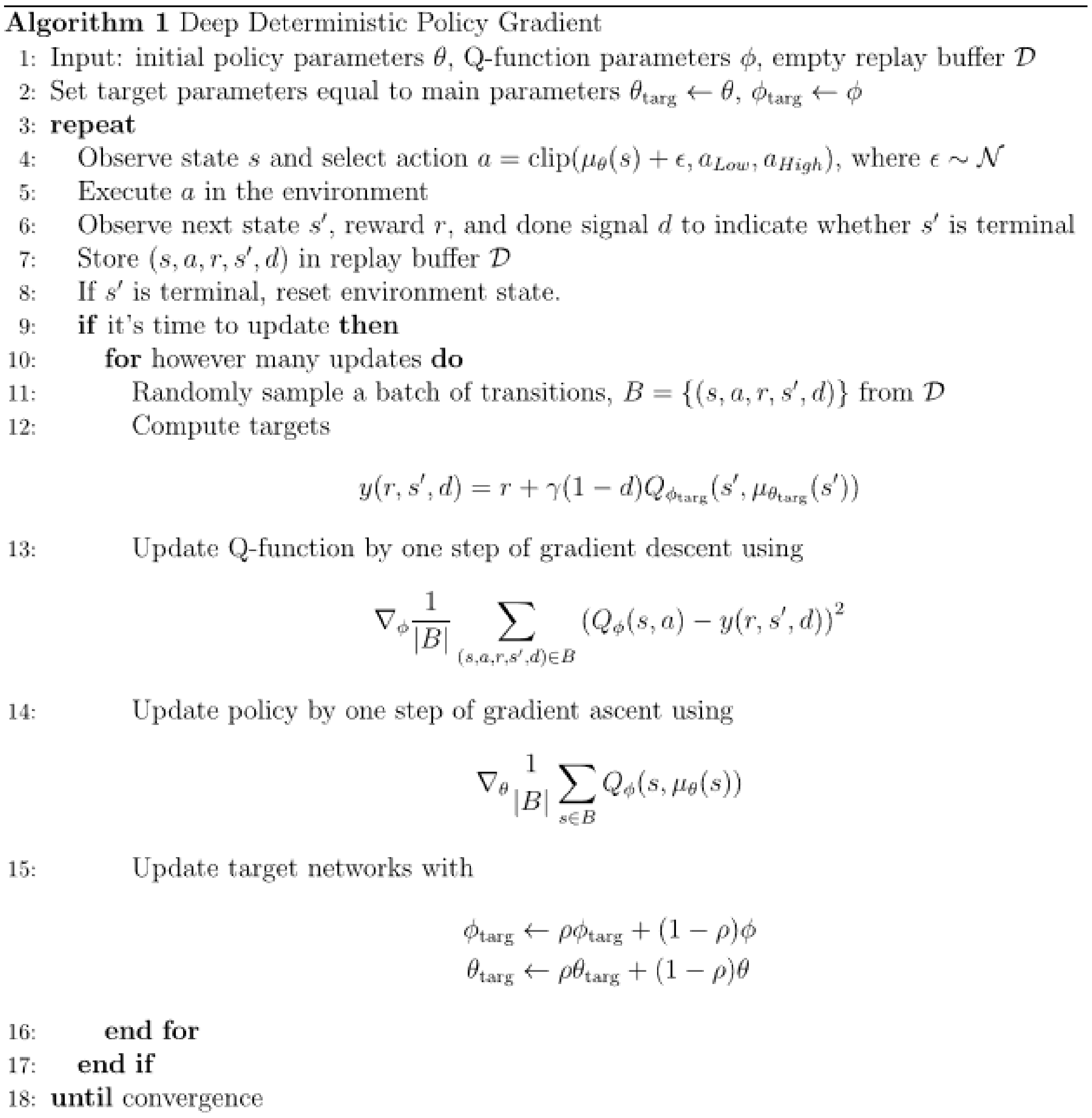
The Q and Policy target network parameters will have a soft update from network after the learning proces.

$$\emptyset_{target} = p * \emptyset_{target} + (1 - p)\ \emptyset$$

where
$\emptyset$ = main network parameter
$\emptyset_{target}$ = target network parameter
p = hyperparameter between 0 and 1

4. Experience Replay Buffers
   As used in many other reinforcement learning algorithms, experience replay buffer will be also used in the DDPG. The replay buffer should be large enough to contain a wide range of experiences such as state, action, reward and next_state. After an episode finishes, we can sample random mini-batches of experience from the replay buffer and then update the Q network and the policy network.

Pseudocode of DDPG



```
Algorithm 1 Deep Deterministic Policy Gradient
1:  Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2:  Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3:  repeat
4:      Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, 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 terminal, reset environment state.
9:      if it's time to update then
10:         for however many updates do
11:             Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:             Compute targets
                    $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:             Update Q-function by one step of gradient descent using
                    $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$
14:             Update policy by one step of gradient ascent using
                    $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:             Update target networks with
                    $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
                    $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$
16:         end for
17:     end if
18: until convergence
```

Pseudocode [5]

Reference

[1] SpaceX CRS-17 Mission
https://www.youtube.com/watch?v=AQFhX5TvP0M

[2] OpenAI Gym LunarLanderContinuous environment
https://gym.openai.com/envs/LunarLanderContinuous-v2/

[3] OpenAI Gym Leaderboard
https://github.com/openai/gym/wiki/Leaderboard#lunarlander-v2

[4] Actor-critic Architecture

https://arxiv.org/pdf/1509.02971.pdf

[5] DDPG Alogrithms
https://spinningup.openai.com/en/latest/algorithms/ddpg.html