Machine Learning Engineer Nanodegree

Capstone Project

Ronin Ho

July 12, 2019

I. Definition

Project Overview

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 explore the environment, making a decision, taking an action and checking how the environment responds and learning from the experiences. This is similar to how human is growing by learning from the past experiences.

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

This SpaceX mission draws my interest to solve the LunarLander environment in OpenAI gym.

Problem Statement

In this project, LunarLander environment in OpenAI gym will be used. This environment provides a lander and a landing pad in a box environment.

I will implement a reinforcement learning agent to learn flying and controlling the lander to land on the landing pad steadily. The agent will observe the lander information such as coordinates, velocity, angular velocity and lander legs status and then take the actions which are the powers of main engine and side engines. Rewards will be given to the agent according to the action its taken on the environment. The rewards can be positive or negative. The agent will keep playing and learning continuously. It is expected that the agent

should learn the gaming rules in enough episodes and find an approach to get the rewards as much as possible.

In this environment, a landing pad is located at coordinates (0,0) while the lander will be located at the top of the environment at the beginning. When lander moves towards the landing pad and successfully landed, the agent will gain rewards. If the lander moves away the landing pad or the lander is crashed, the agent will lose rewards.

The action taken by the agent is the powers of main engine and the side engine. Firing both engines will also lose some rewards.

To get as much as rewards it can, agent should use less fuel to control the lander to land on the landing pad steadily.

Metrics

From the definition of Lunar Lander environment, getting 200 rewards or above in an episode is considered as solving the environment.

In this project, I will define a successfully landing is to get 200 rewards or above in an episode. And to compare my agent performance with the benchmark models, average rewards over last 20 episodes will be calculated and will used to compare their learning progress and results.

II. Analysis

Datasets Exploration and Exploratory Visualization

Data and inputs are not required to prepare in this reinforcement learning project. All the input data will be provided by Lunar lander Environment. The input data is the lander status of the current state. A state contains 8 elements, which are

1. Lander’s x-coordinates
2. Lander’s y-coordinates
3. Lander’s horizontal velocity
4. Lander’s veritcal velocity
5. Lander’s angle
6. Lander’s angular velocity
7. Lander’s left leg contact status
8. Lander’s right leg contact status

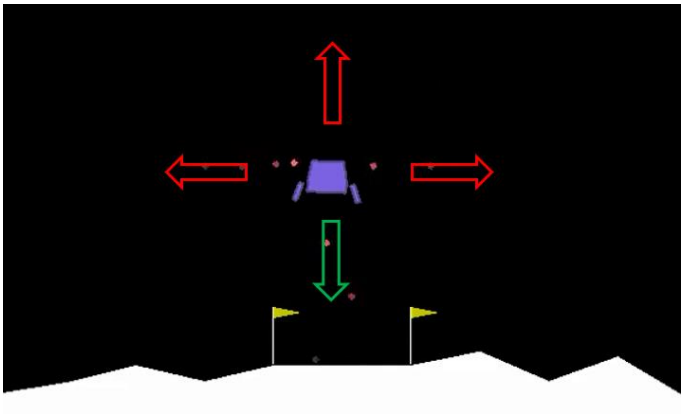
After observing the state, the agent will make the actions which are the power of main engine and the side engine. Both powers are the continuous real values from -1.0 to 1.0.

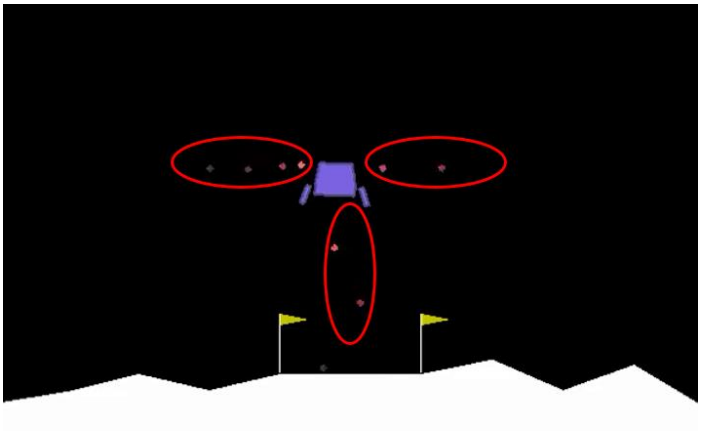
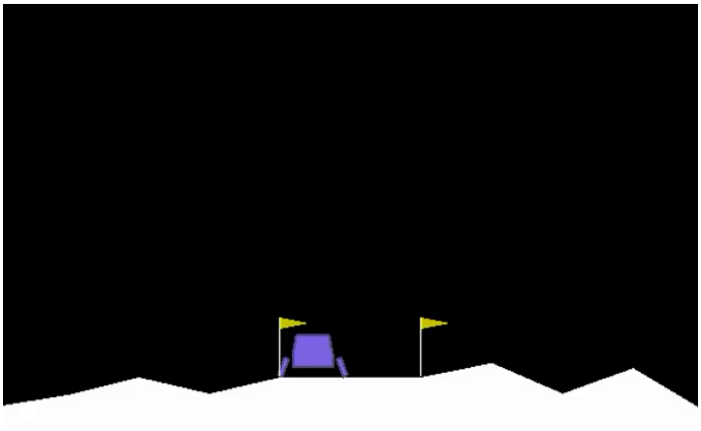
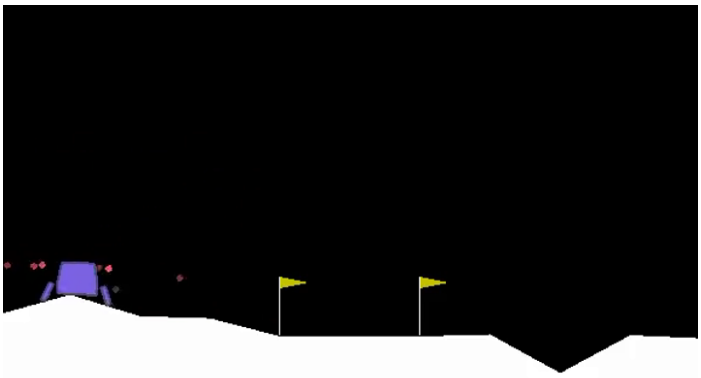
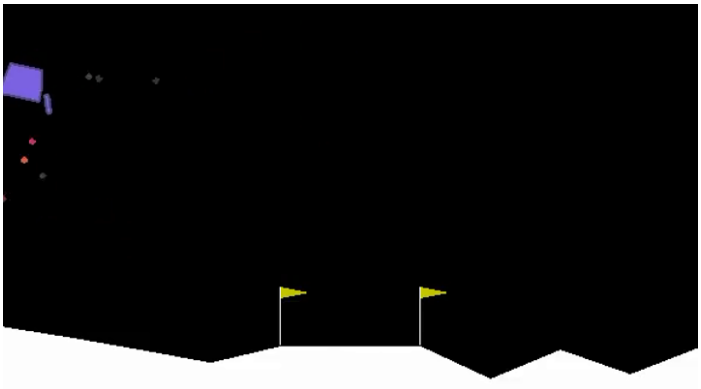
After the agent take the actions on the environment, a reward and the next state information will be returned. Agent can prepare for next decision.

The rewards of the environment will be given in below conditions.

Conditions	Rewards
Lander crashes or move outside the environment	-100
Landers fires main engine	-0.3 per frame
Landers fires sideengine	-0.03 per frame
Lander comes to rest	+100
Lander's leg gound contact (per leg)	+10
Lander moves towards the pad	+ rewards per frame
Lander moves away the pad	- rewards per frame

Scenario

When the agent controls the lander to move away from the landing pad, it will receive negative rewards, see the red arrows. When the agent controls the lander to move closer to the landing pad, it will receive positive rewards, see the green arrow.	

When the agent fires the lander the main engine and side engine, it will also receive negative rewards, see the red circles.	
When the agent controls the lander to land successfully, 100 rewards will be awarded.	
When the agent controls the lander to move outside of the environment or crashed the lander, 100 rewards will be deducted.	 

Algorithms

In this project, I will use Deep Deterministic Policy Gradient (DDPG) Algorithm to implement the agent to work in this environment which has continuous action spaces. DDPG makes use of deep neural networks to compute the state-action values (Q-values) and learn a policy respectively.

When the action spaces are 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 spaces are 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

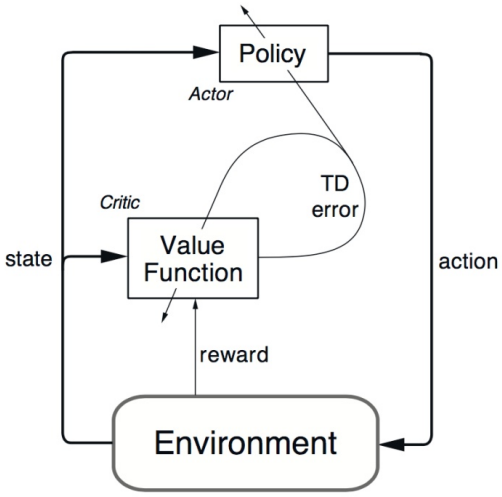
The 2nd Q-value acts as a target value which Q-network should be predicted as closer as possible. By calculating 2 Q-values difference, Q-network is able to know how much error in the prediction. The Q-network can then improve to give more accurate value in next prediction. Also, a gradient of the Q-values 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. Hence, the Policy network can follow the gradient and update its network parameters and make better action.

These techniques make DDPG works in continuous action spaces environment.

Techniques

Below techniques will be used to implement an agent using DDPG alogrithm

1. Actor-Critic architecture



Actor-critic Architecture[3]

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.

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. It may 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. 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.

$$\varnothing_{target} = p * \varnothing_{target} + (1 - p)\, \varnothing$$

where

$\varnothing$ = main network parameter

$\varnothing_{target}$ = 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 experiences from the replay buffer and then update the Q-network and the policy network.

Pseudocode of DDPG

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

Pseudocode [4]

Benchmark

In this project, 2 additional agents will be used as the benchmarks to compare my DDPG agent's learning progress and results.

1. Random Agent
   A simple random agent will be used as a benchmark to show whether my agent is worse or better than acting randomly on the environment. This random agent will simply carry out the actions randomly and does not learn from the observations and rewards.

2. RBF Agent
   This agent makes use of a simple Radial Basis Function (RBF) network[5] to learn to land a lander. One of the typical use case of the RBF network is the function approximation. The RBF network can approximate a continuous function such that it can provide some continuous output values. This use case can be applied to our environment with continuous state spaces and action spaces.

   In this project, RBF Agent will observe the states provided by the environment and generate the actions which are the powers of the main engine and side engine from its RBF network. Before applying the actions to the environment, a Gaussian noise will be added to the agent's actions to allow exploring the environment and actions.

   When the agent is working in an episode, it will record all the experiences including states, actions and rewards for each episode steps. When the episode is finished, the agent will check if the total episode reward is better than the best reward it has ever seen before.

   If yes, the agent will train the RBF network using the states and the actions it saved for this episode. This enables the RBF network approximates the 'best' policy function and try to provide the best actions for a similar state.

   In the next episode, the agent would follow the 'best' policy to explore the environment and try to improve itself.

III. Methodology

Data Preprocessing

In this project, input data is not required to preprocess. All the data is provided by the Lunar Lander environment.

Implementation

1. DDPG Agent
   As mentioned in Techniques section, Actor-Critic architecture, Ornstein-Uhlenbeck Noise, Target Network and Experience Replay Buffers will be used to implement a DDPG Agent. Below will highlight the key implementation of each techniques.

   1.1. Actor Network (Policy network)
   The actor network is used to output the optimal action, e.g. the power of the main engine and the side engine when given a state. The network consists of 4 fully connected layers.
   The input layer takes 8 elements which is the state from the environment.
   1st hidden layer contains 400 nodes with ReLu activation function.

2nd hidden layer contains 300 nodes with ReLu activation function
Then, the final output layer gives 2 output values with Tanh activation function.

The actor network model summary is

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
state_input (InputLayer)     (None, 1, 8)              0
_________________________________________________________________
flatten_1 (Flatten)          (None, 8)                 0
_________________________________________________________________
dense_1 (Dense)              (None, 400)               3600
_________________________________________________________________
activation_1 (Activation)    (None, 400)               0
_________________________________________________________________
dense_2 (Dense)              (None, 300)               120300
_________________________________________________________________
activation_2 (Activation)    (None, 300)               0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 602
_________________________________________________________________
activation_3 (Activation)    (None, 2)                 0
=================================================================
Total params: 124,502
Trainable params: 124,502
Non-trainable params: 0
```

The actor network python code in DDPG.py is below

```python
def build_actor(self):
    state_input = Input(shape=(1,) + self.state_shape, name='state_input')
    state_net = Flatten()(state_input)
    state_net = Dense(400)(state_net)
    state_net = Activation('relu')(state_net)
    state_net = Dense(300)(state_net)
    state_net = Activation('relu')(state_net)
    state_net = Dense(self.action_size)(state_net)
    state_net = Activation('tanh')(state_net)
    actor = Model(inputs=[state_input], outputs=state_net)
    return actor
```

1.2. Actor Learning (Policy Learning)

In actor learning, we want the agent to learn a policy which gives the actions that maximizes the Q-value of a state. In our implementation, the negative of Q-values of the states and the actions from the experiences will be provided to Policy network as a loss function. The larger of the Q-values will lead to the smaller of loss. The framework libraries (Keras and Tensorflow) will calculate the action gradient and update the network parameters to minimize the loss. This can help Policy network to improve the actions decided when a similar state is given.

The python code in DDPG.py is below

```
state_inputs = Input(shape=(1,) + self.state_shape, name='state_inputs')
actions = self.actor_local([state_inputs])
q_values = self.critic_local([state_inputs, actions])
updates = self.actor_local_optimizer.get_updates(params=self.actor_local.trainable_weights,loss=-K.mean(q_values))

#self defined training function
self.actor_train_fn = K.function([state_inputs] + [K.learning_phase()], [self.actor_local(state_inputs)], updates=updates)
```

1.3. Critic Network (Q-network)

The critic network is used to predict the optimal Q-value based on the state and the action. The network consists of 4 fully connected layers.
The input layer takes 8 elements which is the state from the environment.
1st hidden layer contains 400 nodes with ReLu activation function. Then the layer is concatenated with the 2 more nodes which are the actions of the lander.
2nd hidden layer contains 300 nodes with ReLu activation function
Then, the output layer gives 1 output value with Linear activation function.
The output value is the Q-value of the state and action taken by the agent.

The critic network model summary is

```
__________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
state_input (InputLayer)     (None, 1, 8)              0
__________________________________________________________________________________________________
flatten_1 (Flatten)          (None, 8)                 0           state_input[0][0]
__________________________________________________________________________________________________
dense_1 (Dense)              (None, 400)               3600        flatten_1[0][0]
__________________________________________________________________________________________________
activation_1 (Activation)    (None, 400)               0           dense_1[0][0]
__________________________________________________________________________________________________
action_input (InputLayer)    (None, 2)                 0
__________________________________________________________________________________________________
concatenate_1 (Concatenate)  (None, 402)               0           activation_1[0][0]
                                                                   action_input[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)              (None, 300)               120900      concatenate_1[0][0]
__________________________________________________________________________________________________
activation_2 (Activation)    (None, 300)               0           dense_2[0][0]
__________________________________________________________________________________________________
dense_3 (Dense)              (None, 1)                 301         activation_2[0][0]
__________________________________________________________________________________________________
activation_3 (Activation)    (None, 1)                 0           dense_3[0][0]
==================================================================================================
Total params: 124,801
Trainable params: 124,801
Non-trainable params: 0
__________________________________________________________________________________________________
None
```

The critic network python code in DDPG.py is below

```
    def build_critic(self):
        action_input = Input(shape=(self.action_size,), name='action_input')
        state_input = Input(shape=(1,) + self.state_shape, name='state_input')
        net = Flatten()(state_input)
        net = Dense(400)(net)
        net = Activation('relu')(net)
        net = Concatenate()([net, action_input])
        net = Dense(300)(net)
        net = Activation('relu')(net)
        net = Dense(1)(net)
        net = Activation('linear')(net)
        critic = Model(inputs=[state_input, action_input], outputs=net)
        return critic
```

1.4. Critic Learning (Q Learning)

After an episode is finished, the agent will learn from a mini-batch of experiences. Agent will calculate the target Q-value of a visited state and corresponding actions by using the rewards received and the Q-values of next state. The agent will then train and improve the Q-network to predict the Q-value as closer as the Q target value.

By applying Bellman equation, the target Q-value will be calculated as below

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

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

The python code in DDPG.py is below

```
# Get predicted next-state actions and Q values from target models
actions_next = self.actor_target.predict_on_batch(next_states)
Q_targets_next = self.critic_target.predict_on_batch([next_states, actions_next]).flatten()

# Compute Q targets for current states and train critic model (local)
Q_targets = rewards + self.gamma * Q_targets_next * (1 - dones)
Q_targets = Q_targets.reshape(self.batch_size, 1)
self.critic_local.train_on_batch(x=[states, actions], y=Q_targets)
```

1.5. Ornstein-Uhlenbeck Noise

The Ornstein-Uhlenbeck Noise (OU Noise) is added to the action decided by the agent. This can help the agent to explore the combinations of the state and the action values.

The python code in util.py is below

```
class OUNoise:
    def __init__(self, size, mu, theta, sigma):
        """Initialize parameters and noise process."""
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(len(x))
        self.state = x + dx
        return self.state
```

1.6. Target Network Update

In order to prevent training our Actor network and Critic network to chase the moving targets, target networks technique is used and this can stabilize the training progress. After each episode training, soft updates to the target networks of Actor and Critic will be performed.

The python code in DDPG.py is below

```
def soft_update(self, local_model, target_model):
    """Soft update model parameters."""
    local_weights = np.array(local_model.get_weights())
    target_weights = np.array(target_model.get_weights())

    new_weights = self.tau * local_weights + (1 - self.tau) * target_weights
```

1.7. Experience Replay Buffer

The replay buffer is used to store all the experiences such as state, action, reward and next state during the agent learning. After each episode finishes, the agent will sample random mini-batches of experiences from the replay buffer and then update the Q-network and the policy network.

The python code in DDPG.py is below

```
class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.memory = deque(maxlen=buffer_size)  # internal memory (deque)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "reward", "next_st
ate", "done"])

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self, batch_size=64):
        """Randomly sample a batch of experiences from memory."""
        return random.sample(self.memory, k=self.batch_size)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

2. Random Agent

The random agent simply sample the actions from the action spaces in each episode steps.

The python in RandomAgent.py is below

```
class RandomAgent(object):
    def __init__(self, env):
        self.action_space = env.action_space
    def act(self, state):
        return self.action_space.sample()
```

3. RBF Agent

3.1. RBF Network

The RBF network is used to predict the action used by the agent. The network consists of 4 fully connected layers.
The input layer takes 8 elements which is the state from the environment.
1st hidden layer contains 16 nodes with ReLu activation function.
2nd hidden layer contains 16 nodes with RBF activation function
Then, the output layer gives 2 output values.
The output values are actions will be taken by the agent.

The RBF network summary is

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_6 (Flatten)          (None, 8)                 0
_________________________________________________________________
dense_9 (Dense)              (None, 16)                144
_________________________________________________________________
activation_8 (Activation)    (None, 16)                0
_________________________________________________________________
rbf_layer_2 (RBFLayer)       (None, 16)                272
_________________________________________________________________
dense_10 (Dense)             (None, 2)                 34
=================================================================
Total params: 450
Trainable params: 450
Non-trainable params: 0
_________________________________________________________________
None
```

The RBFnetwork python code in RBFAgent.py is below

```
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(RBFLayer(16,betas=self.rbf_beta))
model.add(Dense(self.action_size))
model.compile(optimizer=RMSprop(), loss='mse')
```

3.2. RBF Agent Learning

After an episode is finished, the agent will check if the current episode reward is better than the best reward. If yes, agent will learn from the episode experiences and try to approximate the best actions when similar states are observed.

The python code in RBFAgent.py is below

```
def learn(self):
    if self.episode_reward > self.best_episode_reward:
        self.best_episode_reward = self.episode_reward
        X = np.array(self.memory_state)
        y = np.array(self.memory_action)
        self.actor.fit(X, y,
          batch_size=40,
          epochs=1000,
          verbose=1)
```

Refinement

2 techniques are applied for improvement during the implementation.

1. Normalization of the the input states

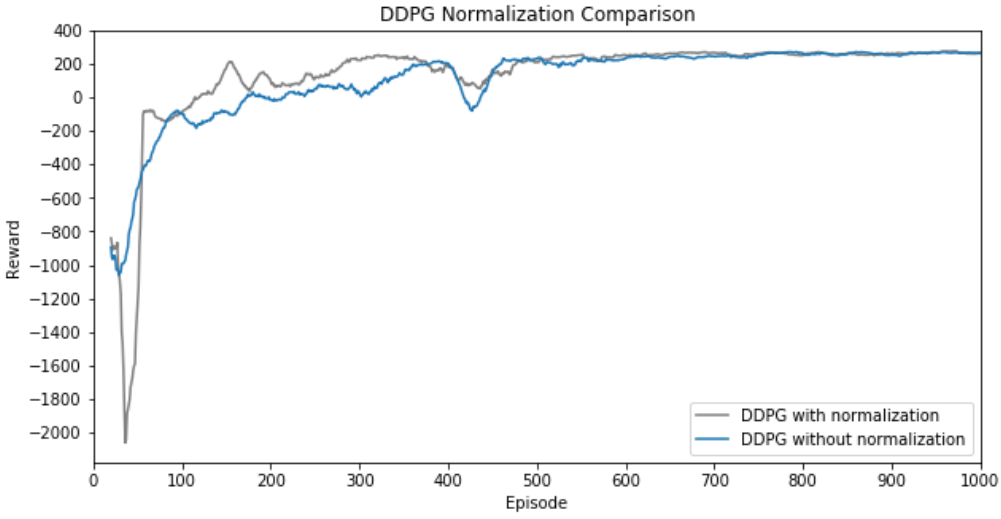
   When I was studying the DDPG algorithm, i read someone proposing applying normalization to the state before learning. In a complex environment and the state elements are not on the same scale, the agent may find difficult in learning.
   In fact, the normalization process is similar what we did apply the MinMaxScaler in sklearn to preprocess the data in previous Udacity projects. This could help boosting the learning progress.

   After the implementation is completed, I compared the learning progress with and without the state normalization. It is found that the progress of them are similar and the state normalization does not improve the progress too much, see below plot.

   I think the reason is the Lunar Lander environment is not complicated enough such that this technique cannot show the improvement.

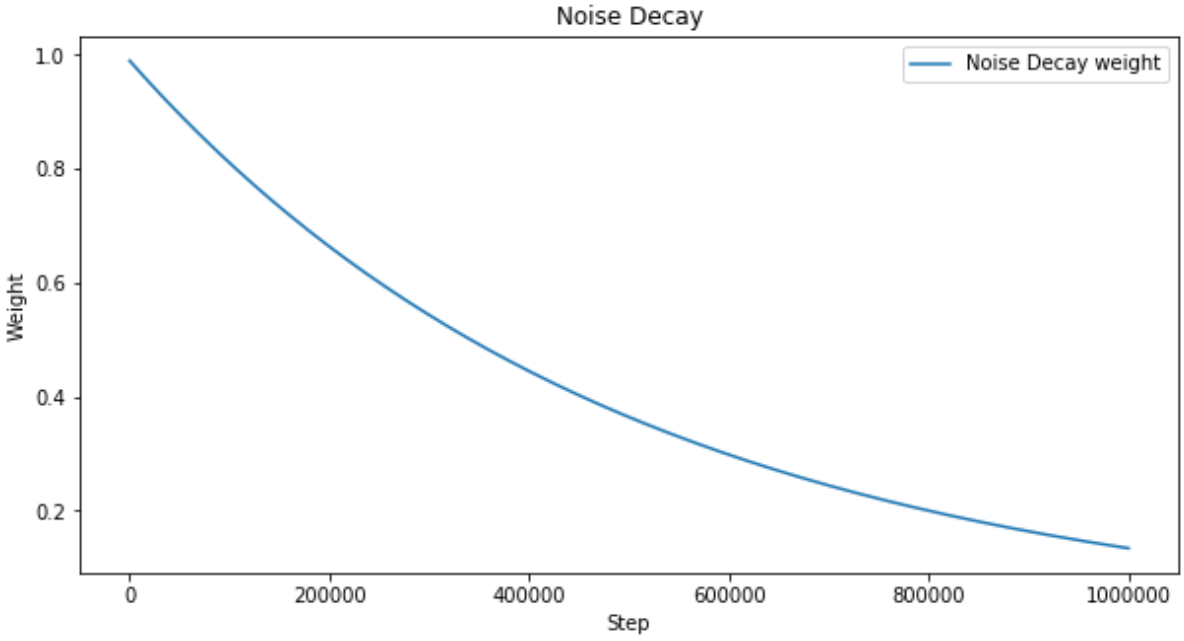
   I would disable the normalization technique in the implementation by default in order to make the agent a little bit simpler.

   DDPG Comparison with and without state normalization



2. Noise Decay
   In order to reduce the exploration rate of the environment in the later part of the training and allow agent to follow the policy to act on the environment, epsilon noise decay technique is used.

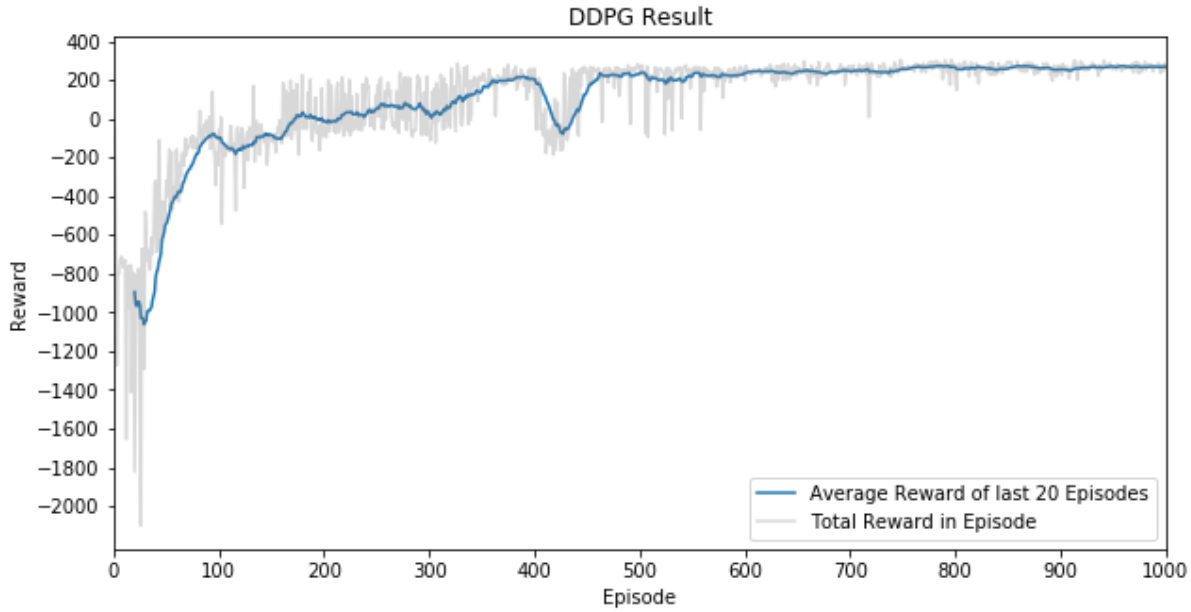
   The noise epsilon used is 0.99
   The noise decay rate is 0.999998



IV. Results

Model Evaluation and Validation

After 1000 episodes of training, DDPG agent has below learning progress.



From above plot, we can see the DDPG agent can keep landing successfully after 600 episodes, which a successful landing is to get 200 rewards or above in an episode defined in Metrics Section.

In the first 400 episodes, DDPG Agent keeps learning and improving itself so the rewards are increasing. Between the 400th and 500th episodes, the agent experienced some volatility in training. But, the agent rectified and converged to an optimal policy.

In the whole training, DDPG Agent can perform 762 successfully landings in 1000 episodes.

After the training, the DDPG Actor and Critic models are saved. And then the model are reloaded to perform 5 more episodes for testing.
Below are the testing result.

Episode number	Rewards
1	248.46
2	262.27
3	243.98
4	259.09
5	279.16

From the above testing result, the DDPG agent can continuously control the lander to land successfully. This proves the agent can solve the environment.

A landing video can be found at https://github.com/ronin-ho/udacity-rl-capstone/blob/master/test_result/DDPG_LunarLanderContinuous.mp4

Justification

In this project, I used 2 benchmark agents to compare my DDPG agent results.
Below are the benchmark agents results and comparsion.

1. Benchmark 1 - Random Agent

From the plot, the Random Agent keeps getting negative rewards. This is he expected result as the agent acts on the environment randomly and does not learn from the experiences.

2. Benchmark 2 - RBF Agent

From the plot, the RBF Agent can sometimes perform successful landing, but it can also get very bad rewards. The overall result is quite volatile. The RBF Agent cannot learn an optimal policy in 1000 episodes.

In the whole training, RBF Agent can perform 68 successfully landings in 1000 episodes.

3. Agents Result Comparison

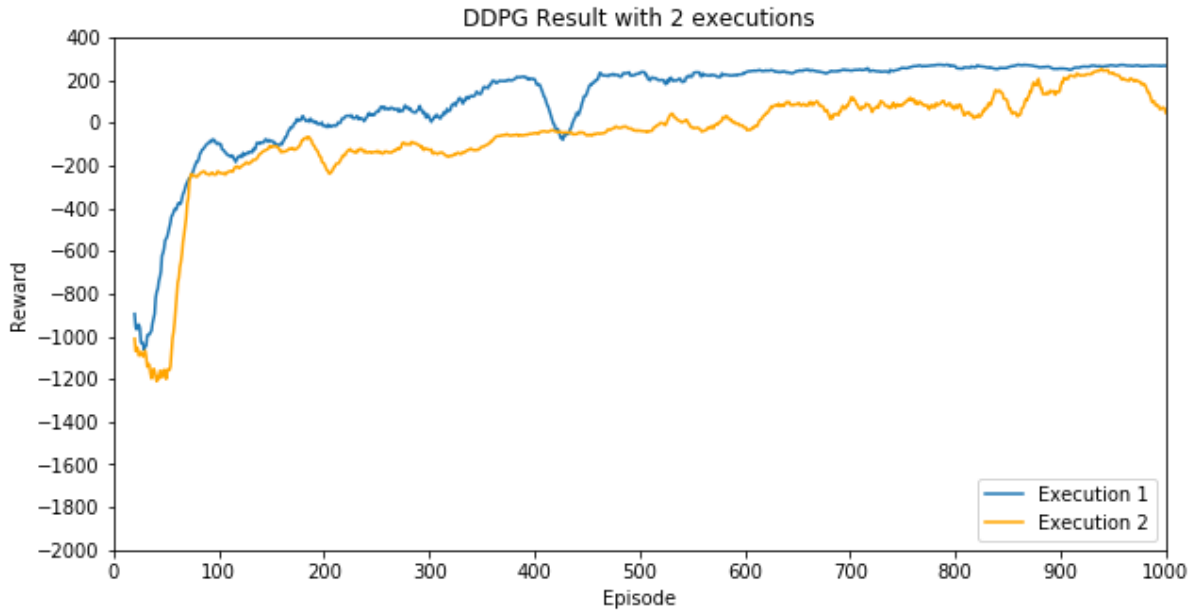
The average reward of last 20 episode from 3 agents are provided in above plot. It is clearly showed that DDPG Agent learns effectively and successfully determined an optimal policy to achieve as much as rewards as possible in 1000 episodes. Regarding the Random Agent and RBF Agent, they are not able to learn an optimal policy in the 1000 episodes.

To conclude, the DDPG Agent can successfully solved the environment and works better than the benchmark agents.

V. Conclusion

Free-Form Visualization

In this section, I would like to highlight even the DDPG Agent implementation and the hyperparameters settings remain the same, the learning progress can be different in a separate training execution.



The above plot shows the average reward of last 20 episode for 2 DDPG Agent trainings. We can see that the agent can learn and improve itself to get higher reward in these trainings. However, the agent can quickly determined an optimal policy in Execution 1 while the agent still needs more episodes to obtain an optimal policy in Execution 2.

The reason of this difference may be related to the randomness of the OU Noise and Experience Replay Buffer.

The OU Noise is somehow generated randomly and is added to the agent predicted actions to explore the environment. If the OU Noise is not generated positively, the agent may not able to receive a positive reward and save it in the Experience Replay Buffer. This may slow down the agent learning progress.
The Experience Replay Buffer also randomly provides a mini-batch of experiences for the agent to learn. If the buffer could not provide enough new or positive experiences to the agent. This may slow down the agent learning progress. But, this issue can be improved by using Priority Experience Replay Buffer, more details will be given in Improvement Section.

Reflection

The process used for this project can be summarized using the following steps

1. Study the DDPG algorithum
2. Research for the benchmark models
3. Implement the DDPG components
4. Test the DDPG agent
5. Tune the hyperparameters
6. Study and include the State Normalizationa and Noise Decay
7. Test the DDPG agent again to see if there is any improvement
8. Implement the benchmark agents
9. Test the benchmark agents
10. Compare the agents results

The most challenging part of this project is the testing of the agent and the tunning of the hyperparameters. In the reinforcement learning, there is no one can guarantee an algorithm and an implementation can lead an agent to learn the rules and determine an optimal policy. I have to give time to agent for exploring and learning the environment. After several hours of the agent's training, I found the agent cannot continuously get the positive episode rewards. I would question the implementation and the hyperparameters settings, e.g. are the networks not complicated enough for learning? or the agent still needs more episodes to train? I have to keep updating them and hope the agent can perform well in next run. However, after several times of testing, I realized that there is a minor bug in the coding and this may attribute to the poor performance of the agent. Debugging, Tuning and Improving in reinforcement learning are the most difficult part in this project.

As the DDPG is a general algorithm to solve the environments with continuous state spaces and action spaces, this project can be reused to solve other environments. However, the actor network and critic network can be adjusted depending the complexity of the environment.

Improvement

To further enhance this project, below techniques can be included when implementing a DDPG agent.

1. Priority Experience Replay Buffer
   As discussed in Free Form Visualization, different learning results can be encountered for the same DDPG agent due to the randomness of the experience replay buffer. If the agent is unlucky that it cannot sample some critical experiences such as the landing step which awards 100 rewards to agent, agent may not able to know landing steadily can maximize the Q-values as well as the rewards. Agent needs to perform more episodes and then realizes this rule.

   To tackle this problem, we can assign a priority to each experiences. The priority can be the absolute difference between the Q-value and target Q-value of an experience, refer the target Q-value calculation in section. The larger of the difference means the higher priority of an experience to be sampled. In fact, the Q-value difference of an experience indicates the agent cannot predict the Q-values correctly and hence the agent does not choose the actions wisely.

   In practice, when an experience is added to the replay buffer, the default priority can be the maximum. When sampling experience batch from the replay buffer, half of the experiences chosen can follow the priority rule while half of the experiences can be selected randomly. After learning, agent should re-calculate the priority of these experiences and update them in the replay buffer. This technique can increase the chance that the agent can learn from the critical experience and speed up the learning.

2. Tricks in Twin Delayed DDPG (TD3)
   Twin Delayed DDPG[6] is an algorithm found in 2018 to improve DDPG. In some of the cases, DDPG would overestimate the Q-values and lead to the error in the policy learning. Below tricks[7] can be able to tackle this issue.

   2.1. Clipped Double Learning
   2 Q-network are used to the optimal Q-values. The critic will use the smaller of the two Q-values to form the Q target value in the learning process.

   2.2. Delayed Policy Learning
   The policy network, policy target network and Q target networks are updated less frequently than the Q-network. The paper recommends one policy update for every two Q-network updates. This can help reducing the volatility in training that normally arises in DDPG because a policy update always changes the target.

VI. 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] Actor-critic Architecture
https://arxiv.org/pdf/1509.02971.pdf

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

[5] Radial Basis Network
https://en.wikipedia.org/wiki/Radial_basis_function_network

[6] Twin Delayed DDPG
https://arxiv.org/abs/1802.09477

[6] Twin Delayed DDPG Tricks
https://spinningup.openai.com/en/latest/algorithms/td3.html

End of Report