Tut6_DDPG

April 26, 2022

[1]:
```
import numpy as np
import gym
from collections import deque
import random

# Ornstein-Ulhenbeck Process
# Taken from #https://github.com/vitchyr/rlkit/blob/master/rlkit/
↪exploration_strategies/ou_strategy.py
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.3,␣
↪min_sigma=0.3, decay_period=100000):
        self.mu           = mu
        self.theta        = theta
        self.sigma        = max_sigma
        self.max_sigma    = max_sigma
        self.min_sigma    = min_sigma
        self.decay_period = decay_period
        self.action_dim   = action_space.shape[0]
        self.low          = action_space.low
        self.high         = action_space.high
        self.reset()

    def reset(self):
        self.state = np.ones(self.action_dim) * self.mu

    def evolve_state(self):
        x  = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(self.
↪action_dim)
        self.state = x + dx
        return self.state

    def get_action(self, action, t=0):
        ou_state = self.evolve_state()
        self.sigma = self.max_sigma - (self.max_sigma - self.min_sigma) * min(1.
↪0, t / self.decay_period)
        return np.clip(action + ou_state, self.low, self.high)
```

```python
# https://github.com/openai/gym/blob/master/gym/core.py
class NormalizedEnv(gym.ActionWrapper):
    """ Wrap action """

    def action(self, action):
        act_k = (self.action_space.high - self.action_space.low)/ 2.
        act_b = (self.action_space.high + self.action_space.low)/ 2.
        return act_k * action + act_b



class Memory:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = deque(maxlen=max_size)

    def push(self, state, action, reward, next_state, done):
        experience = (state, action, np.array([reward]), next_state, done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        reward_batch = []
        next_state_batch = []
        done_batch = []

        batch = random.sample(self.buffer, batch_size)

        for experience in batch:
            state, action, reward, next_state, done = experience
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)

        return state_batch, action_batch, reward_batch, next_state_batch,␣
↪done_batch

    def __len__(self):
        return len(self.buffer)
```

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution across a discrete action space.

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.

Let's create these networks.

[2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd
from torch.autograd import Variable

class Critic(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state, action):
        """
        Params state and actions are torch tensors
        """
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)

        return x

class Actor(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, learning_rate =␣
 ↪3e-4):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state):
        """
        Param state is a torch tensor
        """
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x))
```

```
        return x
```

Now, let's create the DDPG agent. The agent class has two main functions: "get_action" and "update":

- **get_action()**: This function runs a forward pass through the actor network to select a determinisitic action. In the DDPG paper, the authors use Ornstein-Uhlenbeck Process to add noise to the action output (Uhlenbeck & Ornstein, 1930), thereby resulting in exploration in the environment. Class OUNoise (in cell 1) implements this.
- **update()**: This function is used for updating the actor and critic networks, and forms the core of the DDPG algorithm. The replay buffer is first sampled to get a batch of experiences of the form .

The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation. However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value:

For the policy function, our objective is to maximize the expected return. To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter. Keep in mind that the actor (policy) function is differentiable, so we have to apply the chain rule.

But since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch:

We make a copy of the target network parameters and have them slowly track those of the learned networks via "soft updates," as illustrated below:

[22]:
```
import torch
import torch.autograd
import torch.optim as optim
import torch.nn as nn
# from model import *
# from utils import *

class DDPGagent:
    def __init__(self, env, hidden_size=256, actor_learning_rate=1e-4,␣
↪critic_learning_rate=1e-3, gamma=0.99, tau=5e-2, max_memory_size=50000):
        # Params
        self.num_states = env.observation_space.shape[0]
        self.num_actions = env.action_space.shape[0]
        self.gamma = gamma
        self.tau = tau

        # Networks
        self.actor = Actor(self.num_states, hidden_size, self.num_actions)
        self.actor_target = Actor(self.num_states, hidden_size, self.
↪num_actions)
```

```
        self.critic = Critic(self.num_states + self.num_actions, hidden_size,␣
↪self.num_actions)
        self.critic_target = Critic(self.num_states + self.num_actions,␣
↪hidden_size, self.num_actions)


        for target_param, param in zip(self.actor_target.parameters(), self.
↪actor.parameters()):
            target_param.data.copy_(param.data)

        for target_param, param in zip(self.critic_target.parameters(), self.
↪critic.parameters()):
            target_param.data.copy_(param.data)

        # Training
        self.memory = Memory(max_memory_size)
        self.critic_criterion  = nn.MSELoss()
        self.actor_optimizer  = optim.Adam(self.actor.parameters(),␣
↪lr=actor_learning_rate)
        self.critic_optimizer = optim.Adam(self.critic.parameters(),␣
↪lr=critic_learning_rate)

    def get_action(self, state):
        state = Variable(torch.from_numpy(state).float().unsqueeze(0))
        action = self.actor.forward(state)
        action = action.detach().numpy()[0,0]
        return action

    def update(self, batch_size):
        states, actions, rewards, next_states, _ = self.memory.
↪sample(batch_size)
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)

        # Implement critic loss
        next_actions = self.actor_target(next_states)

        next_state_action_values = self.critic_target(next_states, next_actions.
↪detach())

        expected_values = rewards +   self.gamma * next_state_action_values

        self.critic_optimizer.zero_grad()
```

```
        state_actions = self.critic(states, actions)

        critic_loss = self.critic_criterion(state_actions, expected_values.
↪detach())

        critic_loss.backward()

        self.critic_optimizer.step()



        # Implement actor loss

        self.actor_optimizer.zero_grad()

        actor_loss = -self.critic(states, self.actor(states)).mean()

        actor_loss.backward()

        # Update actor and critic networks

        # critic_loss.backward()

        # self.critic_optimizer.step()

        # actor_loss.backward()

        self.actor_optimizer.step()



        # update target networks

        for target_param, param in zip(self.actor_target.parameters(), self.
↪actor.parameters()):
            target_param.data.copy_(param.data * \
                  self.tau + target_param.data * (1.0 - self.tau))

        for target_param, param in zip(self.critic_target.parameters(), self.
↪critic.parameters()):
            target_param.data.copy_(param.data * \
                    self.tau + target_param.data * (1.0 - self.tau))
```

Putting it all together: DDPG in action.

The main function below runs 100 episodes of DDPG on the "Pendulum-v0" environment of OpenAI gym. This is the inverted pendulum swingup problem, a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing

it up so it stays upright.

Each episode is for a maximum of 500 timesteps. At each step, the agent chooses an action, updates its parameters according to the DDPG algorithm and moves to the next state, repeating this process till the end of the episode.

The DDPG algorithm is as follows:

[11]:
```
import sys
import gym
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

env = NormalizedEnv(gym.make("Pendulum-v0"))

agent = DDPGagent(env)
noise = OUNoise(env.action_space)
batch_size = 128
rewards = []
avg_rewards = []

for episode in range(50):
    state = env.reset()
    noise.reset()
    episode_reward = 0

    for step in range(500):
        action = agent.get_action(state)
        #Add noise to action

        action = noise.get_action(action, t = step)
        new_state, reward, done, _ = env.step(action)
        agent.memory.push(state, action, reward, new_state, done)

        if len(agent.memory) > batch_size:
            agent.update(batch_size)

        state = new_state
        episode_reward += reward

        if done:
            sys.stdout.write("episode: {}, reward: {}, average _reward: {} \n".
↪format(episode, np.round(episode_reward, decimals=2), np.mean(rewards[-10:
↪])))
            break

    rewards.append(episode_reward)
```

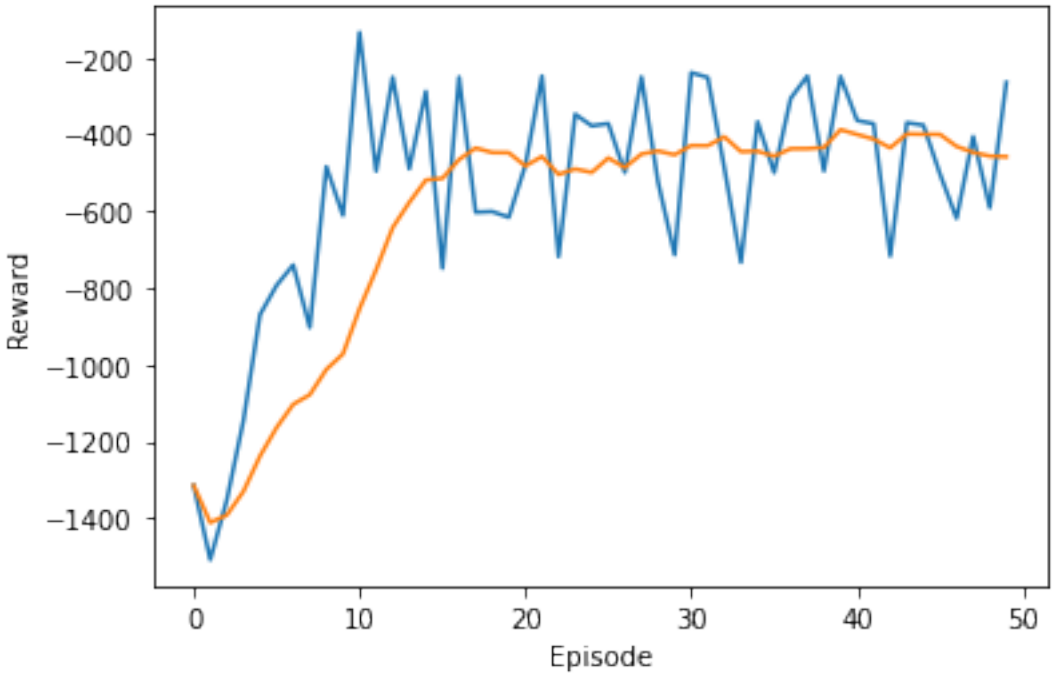
```
    avg_rewards.append(np.mean(rewards[-10:]))

plt.plot(rewards)
plt.plot(avg_rewards)
plt.plot()
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:3441:
RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:189:
RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

```
episode: 0, reward: -1314.25, average _reward: nan
episode: 1, reward: -1507.63, average _reward: -1314.2470686959234
episode: 2, reward: -1351.59, average _reward: -1410.9369995851207
episode: 3, reward: -1145.16, average _reward: -1391.1547410766161
episode: 4, reward: -869.21, average _reward: -1329.6550152419113
episode: 5, reward: -793.03, average _reward: -1237.5652622920709
episode: 6, reward: -739.94, average _reward: -1163.4762369013636
episode: 7, reward: -902.88, average _reward: -1102.9715784600724
episode: 8, reward: -484.05, average _reward: -1077.9605426737621
episode: 9, reward: -610.9, average _reward: -1011.9699370742458
episode: 10, reward: -134.21, average _reward: -971.8628884526245
episode: 11, reward: -495.42, average _reward: -853.8591597014404
episode: 12, reward: -251.11, average _reward: -752.6381525322578
episode: 13, reward: -489.62, average _reward: -642.5901841791454
episode: 14, reward: -288.0, average _reward: -577.0368814703078
episode: 15, reward: -748.69, average _reward: -518.9162112805998
episode: 16, reward: -250.78, average _reward: -514.4817471879762
episode: 17, reward: -602.8, average _reward: -465.56488950050306
episode: 18, reward: -600.99, average _reward: -435.5564341102765
episode: 19, reward: -615.87, average _reward: -447.25049816125
episode: 20, reward: -480.25, average _reward: -447.7473582227405
episode: 21, reward: -247.91, average _reward: -482.35136346762886
episode: 22, reward: -718.3, average _reward: -457.60065951869535
episode: 23, reward: -347.37, average _reward: -504.3197743693448
episode: 24, reward: -377.74, average _reward: -490.09417565517805
episode: 25, reward: -371.8, average _reward: -499.06816244976443
episode: 26, reward: -499.27, average _reward: -461.3796720540715
episode: 27, reward: -250.31, average _reward: -486.22966692255693
episode: 28, reward: -523.98, average _reward: -450.98094131530945
episode: 29, reward: -713.58, average _reward: -443.2807167063165
episode: 30, reward: -239.16, average _reward: -453.0521391685744
episode: 31, reward: -251.82, average _reward: -428.9432482643254
```

```
episode: 32, reward: -487.95, average _reward: -429.3343241611441
episode: 33, reward: -733.16, average _reward: -406.299189748201
episode: 34, reward: -367.28, average _reward: -444.8785358628499
episode: 35, reward: -499.5, average _reward: -443.8323524672493
episode: 36, reward: -305.89, average _reward: -456.6022081255345
episode: 37, reward: -248.18, average _reward: -437.2641381823907
episode: 38, reward: -495.1, average _reward: -437.05077756677093
episode: 39, reward: -249.04, average _reward: -434.16203397034394
episode: 40, reward: -363.67, average _reward: -387.70755814489803
episode: 41, reward: -372.32, average _reward: -400.1580785686008
episode: 42, reward: -716.75, average _reward: -412.20779815424623
episode: 43, reward: -369.6, average _reward: -435.0879353855369
episode: 44, reward: -376.0, average _reward: -398.7320446910624
episode: 45, reward: -503.08, average _reward: -399.6047416613104
episode: 46, reward: -618.64, average _reward: -399.962465170975
episode: 47, reward: -405.46, average _reward: -431.236803231054
episode: 48, reward: -590.88, average _reward: -446.9649647416444
episode: 49, reward: -264.6, average _reward: -456.5436006556914
```



1 IGNORE

[ ]:
```
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
```

[12]:
```
# Run this only if you are using Google Colab
from google.colab import drive
import os

drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
!jupyter nbconvert --to pdf /content/drive/MyDrive/6Tut/Tut6_DDPG.ipynb
```

[ ]:
```

```