me19b190-tut6-ddpg

March 23, 2023

[ ]:
```
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 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.

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

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 **<states, actions, rewards, next_states>**.

The value network is updated using the Bellman equation, similar to Q-learning. 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 target Q value and the predicted Q value:

For the policy function, our objective is to maximize the expected return. To calculate the policy gradient, we take the derivative of the objective function with respect to the policy parameter. For this, we use the chain rule.

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:

[ ]:
```
import torch
import torch.optim as optim
import torch.nn as nn

class DDPGagent:
    def __init__(self, env, hidden_size=256, actor_learning_rate=1e-4,␣
 ↪critic_learning_rate=1e-3, gamma=0.99, tau=1e-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 = torch.FloatTensor(state).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 and update critic
        next_actions = self.actor_target(next_states)   #max_a Q(s,a) is␣
↪approximated by Q(s,policy network(s))
```

```
        Q_targets  = rewards+(self.gamma*self.
↪critic_target(next_states,next_actions.detach()))  #getting the TD target
        Q_expected = self.critic(states,actions)                  #finding the␣
↪action values of the current state action pairs
        loss = self.critic_criterion(Q_expected,Q_targets)      #MSE loss␣
↪caluclation
        self.critic_optimizer.zero_grad()                       #setting␣
↪previous gradients to zero (so they do not accumulate)
        loss.backward()                                         #caluclating␣
↪the gradients
        self.critic_optimizer.step()                            #updating the␣
↪weights of the critic network

        # Implement actor loss and update actor
        Q_expected = self.critic(states,self.actor(states)) #Calculate Q(s,a)␣
↪for all observed states and actions(determined by policy network)
        loss_actor = -1*Q_expected.mean()    #maximize the avergae of state␣
↪action values at all observed states, where the action taken in a state is␣
↪determined by the policy network
        self.actor_optimizer.zero_grad()     #setting previous gradients to␣
↪zero (so they do not accumulate)
        loss_actor.backward()                #caluclate the gradients
        self.actor_optimizer.step()          #update the weights of the actor␣
↪network

        # update target networks
        for target_param, param in zip(self.actor_target.parameters(), self.
↪actor.parameters()): #polyak averaging for actor network
            target_param.data.copy_(self.tau*param.data + (1-self.
↪tau)*target_param.data)

        for target_param, param in zip(self.critic_target.parameters(), self.
↪critic.parameters()): #polyak averaging for critic network
            target_param.data.copy_(self.tau*param.data + (1-self.
↪tau)*target_param.data)
```

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

The DDPG algorithm is as follows:

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

# For more info on the Pendulum environment, check out https://www.gymlibrary.
 ↪dev/environments/classic_control/pendulum/
env = NormalizedEnv(gym.make("Pendulum-v1"))

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

for episode in range(100):
    state = env.reset()
    noise.reset()
    episode_reward = 0

    for step in range(200):
        action = agent.get_action(state)
        #Add noise to action

        action = noise.get_action(action,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()
```

```
episode: 0, reward: -1588.14, average _reward: nan
episode: 1, reward: -1459.77, average _reward: -1588.1422492195722
episode: 2, reward: -1283.25, average _reward: -1523.9554310370295
episode: 3, reward: -1477.3, average _reward: -1443.7215880907181
episode: 4, reward: -1378.04, average _reward: -1452.1158469990503
episode: 5, reward: -914.8, average _reward: -1437.3014914661749
episode: 6, reward: -663.32, average _reward: -1350.217975553251
episode: 7, reward: -709.88, average _reward: -1252.0896316676567
episode: 8, reward: -513.76, average _reward: -1184.3134771441778
episode: 9, reward: -258.36, average _reward: -1109.8078224074243
episode: 10, reward: -508.01, average _reward: -1024.6627672828656
episode: 11, reward: -1060.99, average _reward: -916.6497534800416
episode: 12, reward: -646.45, average _reward: -876.7720457887093
episode: 13, reward: -115.99, average _reward: -813.0917372409315
episode: 14, reward: -767.78, average _reward: -676.9612045002473
episode: 15, reward: -486.88, average _reward: -615.934702781707
episode: 16, reward: -509.56, average _reward: -573.1428004566616
episode: 17, reward: -366.83, average _reward: -557.7673152461236
episode: 18, reward: -132.4, average _reward: -523.4620128170526
episode: 19, reward: -253.48, average _reward: -485.32531508669217
episode: 20, reward: -483.64, average _reward: -484.83723702031955
episode: 21, reward: -495.04, average _reward: -482.4000033212882
episode: 22, reward: -253.95, average _reward: -425.8046677909295
episode: 23, reward: -349.15, average _reward: -386.554101490731
episode: 24, reward: -292.0, average _reward: -409.8698181867447
episode: 25, reward: -367.13, average _reward: -362.29231340039377
episode: 26, reward: -381.36, average _reward: -350.31729776820674
episode: 27, reward: -252.26, average _reward: -337.49730023850253
episode: 28, reward: -254.37, average _reward: -326.0408247358215
episode: 29, reward: -492.46, average _reward: -338.23822127296177
episode: 30, reward: -383.76, average _reward: -362.13643840358145
episode: 31, reward: -511.41, average _reward: -352.1481477536339
episode: 32, reward: -493.61, average _reward: -353.78500181056364
episode: 33, reward: -782.21, average _reward: -377.7513454309226
episode: 34, reward: -375.53, average _reward: -421.0568832863344
episode: 35, reward: -557.17, average _reward: -429.4092229236559
episode: 36, reward: -512.27, average _reward: -448.4127027167222
episode: 37, reward: -504.08, average _reward: -461.50279194324423
episode: 38, reward: -387.21, average _reward: -486.684560272151
episode: 39, reward: -744.68, average _reward: -499.96866916494446
episode: 40, reward: -380.3, average _reward: -525.190874425402
```

```
episode: 41, reward: -375.88, average _reward: -524.8455671860278
episode: 42, reward: -382.42, average _reward: -511.2933076547464
episode: 43, reward: -262.99, average _reward: -500.17493751531674
episode: 44, reward: -815.28, average _reward: -448.25348092858746
episode: 45, reward: -378.79, average _reward: -492.2290549452594
episode: 46, reward: -610.95, average _reward: -474.3913680920964
episode: 47, reward: -505.15, average _reward: -484.2602184985928
episode: 48, reward: -591.42, average _reward: -484.366966951745
episode: 49, reward: -495.35, average _reward: -504.7883488747636
episode: 50, reward: -385.98, average _reward: -479.85526075743917
episode: 51, reward: -255.85, average _reward: -480.4223899691953
episode: 52, reward: -373.28, average _reward: -468.4184897571301
episode: 53, reward: -254.21, average _reward: -467.5038841191458
episode: 54, reward: -247.97, average _reward: -466.62565058659027
episode: 55, reward: -5.2, average _reward: -409.8943974955884
episode: 56, reward: -261.27, average _reward: -372.53573372729016
episode: 57, reward: -486.35, average _reward: -337.5670226317737
episode: 58, reward: -382.66, average _reward: -335.68749493755337
episode: 59, reward: -497.84, average _reward: -314.81090918777704
episode: 60, reward: -501.05, average _reward: -315.0600657535914
episode: 61, reward: -631.82, average _reward: -326.56734650856924
episode: 62, reward: -492.96, average _reward: -364.164655108485
episode: 63, reward: -499.98, average _reward: -376.13235436054345
episode: 64, reward: -375.28, average _reward: -400.709872937772
episode: 65, reward: -614.07, average _reward: -413.4409247031202
episode: 66, reward: -632.35, average _reward: -474.327563133425
episode: 67, reward: -272.14, average _reward: -511.436121436337
episode: 68, reward: -608.39, average _reward: -490.015106755808
episode: 69, reward: -505.22, average _reward: -512.5882447808756
episode: 70, reward: -500.89, average _reward: -513.3257280716988
episode: 71, reward: -589.23, average _reward: -513.30953333796
episode: 72, reward: -379.14, average _reward: -509.0509425141669
episode: 73, reward: -372.72, average _reward: -497.6693115850288
episode: 74, reward: -135.62, average _reward: -484.9427547007663
episode: 75, reward: -255.28, average _reward: -460.9766339885967
episode: 76, reward: -376.1, average _reward: -425.0977443482191
episode: 77, reward: -133.07, average _reward: -399.47268751260196
episode: 78, reward: -498.24, average _reward: -385.5654754918589
episode: 79, reward: -630.75, average _reward: -374.5500569102573
episode: 80, reward: -491.23, average _reward: -387.10310871239864
episode: 81, reward: -379.21, average _reward: -386.1378705889455
episode: 82, reward: -376.28, average _reward: -365.13558511635404
episode: 83, reward: -485.52, average _reward: -364.8499715079623
episode: 84, reward: -580.19, average _reward: -376.12982494554274
episode: 85, reward: -590.21, average _reward: -420.5864817306498
episode: 86, reward: -378.79, average _reward: -454.0793507080507
episode: 87, reward: -376.79, average _reward: -454.34839420882474
episode: 88, reward: -501.41, average _reward: -478.7203297643765
```

```
episode: 89, reward: -512.62, average _reward: -479.03772015165293
episode: 90, reward: -495.93, average _reward: -467.2250930688739
episode: 91, reward: -613.57, average _reward: -467.69505479581994
episode: 92, reward: -618.79, average _reward: -491.1307569556414
episode: 93, reward: -490.11, average _reward: -515.3810009127086
episode: 94, reward: -494.89, average _reward: -515.8398757470143
episode: 95, reward: -499.12, average _reward: -507.3097700703958
episode: 96, reward: -421.06, average _reward: -498.2010629156891
episode: 97, reward: -501.4, average _reward: -502.42756892946943
episode: 98, reward: -373.84, average _reward: -514.8883648174158
episode: 99, reward: -369.46, average _reward: -502.13127185868905
```