ECE433/COS435 Introduction to RL
Assignment 6: Actor-critic Algorithms for continuous action space: DDPG & TD3
Spring 2025

Fill me in

Your name here.

Due April 4, 2025

Collaborators

Fill me in

Please fill in the names and NetIDs of your collaborators in this section.

Instructions

Writeups should be typesetted in Latex and submitted as PDFs. You can work with whatever tool you like for the code, but **please submit the asked-for snippet and answer in the solutions box as part of your writeup. We will only be grading your write-up.**

In this assignment, we will introduce two modern RL algorithms that aim to tackle MDPs with continuous action space: Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3).

The workflow for the rest of this assignment is as follows: we will initially implement DDPG algorithms, and then evaluate their performance on the Gym environment MountainCarContinuous. The implementation for TD3 is directly related to DDPG. By studying these two algorithms, you will gain insight into reinforcement learning in continuous action spaces and become familiar with various techniques in RL.

Question 1. Deep Deterministic Policy Gradient(DDPG)

Question 1.a

First, you want to build your Actor_DDPG and Critic_DDPG networks. Paste your class below:

Solution

```
class Actor_DDPG(nn.Module):
  def __init__(self, state_dim: int, action_dim: int, max_action: torch.tensor):
    super(Actor_DDPG, self).__init__()
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

    self.max_action = max_action

  def forward(self, state):
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################
```

```
class Critic_DDPG(nn.Module):
  def __init__(self, state_dim: int, action_dim: int):
    super(Critic_DDPG, self).__init__()
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

  def forward(self, state: torch.Tensor, action: torch.Tensor)->torch.Tensor:
    sa = torch.cat([state, action], 1)
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################
```

Question 1.b

Now we are ready to construct a DDPG trainer! In the following function you will need to: (1) Calculate the TD value using target_Q network and update the critic; (2) Calculate the deterministic policy gradient and update the actor; (3) Update the target networks.

Paste below:

Solution

```python
def train(self, replay_buffer, batch_size=64):
    # Sample from replay buffer
    state, action, next_state, reward, not_done = replay_buffer.sample(batch_size)

    # Compute the target_Q value with critic_target for the batch

    ###########################
    # YOUR IMPLEMENTATION HERE #
    pass
    ###########################

    # Get current Q estimate
    current_Q = self.critic(state, action)

    # Compute critic loss
    ###########################
    # YOUR IMPLEMENTATION HERE #
    critic_loss =  None
    ###########################

    # Optimize the critic
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # Compute actor loss
    ###########################
    # YOUR IMPLEMENTATION HERE #
    actor_loss = None
    ###########################

    # Optimize the actor
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

    # Update the target models
    for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
      ###########################
      # YOUR IMPLEMENTATION HERE #
      new_target_params = None
      target_param.data.copy_(new_target_params)
      ###########################

    for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
      ###########################
      # YOUR IMPLEMENTATION HERE #
```

```
        new_target_params = None
        target_param.data.copy_(new_target_params)
        ############################
```

Question 1.c

Time to see how it works. We would expect a reward that converges to around 90. The estimated wall time for running the whole process is around $10-20$ minutes, and you should be able to see a large positive reward at around 8×10^4 timesteps. If the initialization is unsuccessful, which could result in the reward being stuck at around 0, try restarting the main function or debugging your trainer.

Note: Even if your implementation has no errors, its performance may not be perfect every time; this is acceptable because it involves randomness. Paste your code for drawing training curve below, also provide the best training curve you can get.

Solution

1. Paste the code for plotting the training curve below:

```
import matplotlib.pyplot as plt

############################
# YOUR IMPLEMENTATION HERE #
pass
############################
```

2. Training curve below:

Question 2. Twin Delayed DDPG (TD3)

Question 2.a

First, we will implement the actor and critic network for TD3. For the actor and every critic (we need to maintain an additional critic), please make sure it has the same structure as the one in the previous DDPG question so that we can conduct an ablation study.

Paste your class below:

Solution

```
class Actor_TD3(nn.Module):
  def __init__(self, state_dim: int, action_dim: int, max_action:
    float):
    super(Actor_TD3, self).__init__()
    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
    self.max_action = max_action
```

```python


  def forward(self, state):
    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
```

Solution

```python
class Critic_TD3(nn.Module):
  def __init__(self, state_dim: int, action_dim: int):
    super(Critic_TD3, self).__init__()
        # Q1 architecture
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

    # Q2 architecture
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################

  def forward(self, state: torch.Tensor, action: torch.Tensor) ->
   Tuple[torch.Tensor, torch.Tensor]:
    sa = torch.cat([state, action], 1)

    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################
    return q1, q2

  # Implement a function that returns only Q1, which is helpful when
   calculating actor loss
  def Q1(self, state: torch.Tensor, action: torch.Tensor) -> torch.
   Tensor:
    sa = torch.cat([state, action], 1)
    ############################
    # YOUR IMPLEMENTATION HERE #
    pass
    ############################
    return q1
```

Question 2.b

Now let's implement the TD3 trainer!

Solution

```python
def train(self, replay_buffer, batch_size=256):
    self.total_it += 1

    # Sample replay buffer
    state, action, next_state, reward, not_done = replay_buffer.
sample(batch_size)

    with torch.no_grad():
      # 1. Select action according to policy and add clipped noise.
      ############################
      # YOUR IMPLEMENTATION HERE #
      pass
      ############################


      # Compute the target Q value
      target_Q1, target_Q2 = self.critic_target(next_state,
next_action)

      # 2. Compute the target_Q here
      ############################
      # YOUR IMPLEMENTATION HERE #
      pass
      ############################

    # Get current Q estimates
    current_Q1, current_Q2 = self.critic(state, action)

    # Compute critic loss
    critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(
current_Q2, target_Q)

    # Optimize the critic
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()

    # Delayed policy updates
    if self.total_it % self.policy_freq == 0:

      # Compute actor loss
      ############################
      # YOUR IMPLEMENTATION HERE #
      pass
      ############################

      # Optimize the actor
      self.actor_optimizer.zero_grad()
      actor_loss.backward()
      self.actor_optimizer.step()
```

```python

        # Update the frozen target models using weighted mean
        for param, target_param in zip(self.critic.parameters(), self.
    critic_target.parameters()):
            ###########################
            # YOUR IMPLEMENTATION HERE #
            new_target_params = None
            target_param.data.copy_(new_target_params)
            ###########################

        for param, target_param in zip(self.actor.parameters(), self.
    actor_target.parameters()):
            ###########################
            # YOUR IMPLEMENTATION HERE #
            new_target_params = None
            target_param.data.copy_(new_target_params)
            ###########################
```

Question 2.c

Time to see how it works. Plot your reward v.s. training_episode curve.

Note: Even if your implementation has no errors, its performance may not be perfect every time; this is acceptable because it involves randomness. Paste your code for drawing training curve below, also provide the best training curve you can get.

Solution

1. Paste the code for plotting the training curve below:

```python
import matplotlib.pyplot as plt

###########################
# YOUR IMPLEMENTATION HERE #
pass
###########################
```

2. Training curve below: