Bonus Point Assignment II

Submission Evaluation Group 40

Overview

Question	Points
A2	14/14
B1	4/4
B2	13/14
C1	12/12
Total	43/44
Bonus points	6/6

6 bonus points in this assignment correspond to 5% of bonus for the final exam.
Below follows the detailed evaluation of your submission. Red boxes are errors which deducted points. In case of unittest failures, code is given with which you can reproduce the error. Yellow boxes denote comments for your information.

A2

a) Defining the Policy

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.fully_connected_layer = nn.Sequential(
            nn.Linear(4, 128), nn.ReLU(), nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, 2)
        )
        self.softmax_layer = nn.Softmax(1)

    def forward(self, x):
        x = self.fully_connected_layer(x)
        x = self.softmax_layer(x)
        return x
```

ok

Network uses 17,410 weights.

b) Action Sampling

```
def sample_action(probs):
    action_distribution = torch.distributions.Categorical(probs=probs)
    action = action_distribution.sample()
    log_prob = action_distribution.log_prob(action)
    return action, log_prob
```

ok

c) Estimate Return

```
def estimate_return(rewards, gamma=0.99):
    returns = []
    reward = 0
    for r in reversed(rewards):
        reward = reward * gamma + r
        returns.insert(0, reward)
    mean = np.mean(returns)
    std = np.std(returns)
    returns = (returns - mean) / std
    return returns
```

ok

d) Training Loop

ok

Average validation reward: 500.0
Average test reward: 500.0

B1

a) Implement Replay Buffer

```
def add(self, state, action, reward, next_state, is_terminal):
    self.state_memory[self.ptr] = state
    self.next_state_memory[self.ptr] = next_state
    self.action_memory[self.ptr] = action
    self.reward_memory[self.ptr] = reward
    self.terminal_memory[self.ptr] = is_terminal
    self.mem_cntr = self.mem_cntr + 1 if not self.is_filled() else self.mem_size
    self.ptr = (self.ptr + 1) % self.mem_size
```

ok

```python
def sample_batch(self, batch_size):
    index = np.random.choice(range(self.mem_cntr), size=batch_size, replace=False)
    states = self.state_memory[index]
    actions = self.action_memory[index]
    rewards = self.reward_memory[index]
    next_states = self.next_state_memory[index]
    is_terminal = self.terminal_memory[index]
    return states, actions, rewards, next_states, is_terminal
```

ok

b) Fill replay buffer

```python
state = env.reset()
while not buffer.is_filled():
    action = env.action_space.sample()
    next_state, reward, done, _ = env.step(action)
    buffer.add(state, action, reward, next_state, done)
    if done:
        state = env.reset()
    else:
        state = next_state
```

ok

B2

a) Define Q-Network

```python
class DeepQNetwork(nn.Module):
    def __init__(self):
        super(DeepQNetwork, self).__init__()
        self.fully_connected_layer = nn.Sequential(
            nn.Linear(4, 128), nn.ReLU(), nn.Linear(128, 128), nn.ReLU(),
            nn.Linear(128, 2)
        )

    def forward(self, state):
        Q = self.fully_connected_layer(state)
        return Q
```

ok

Network uses 17,410 weights.

b) ϵ-Greedy

```python
def epsilon_greedy(state, q_network, epsilon=0.05):
    state = torch.tensor(state, dtype=torch.float32)
    if torch.rand(1) < epsilon:
        action = env.action_space.sample()
    else:
        action = q_network(state).argmax().item()
    return action
```

AssertionError

```python
class DummyModule(nn.Module):
    def forward(self, state):
        return torch.FloatTensor([-1, -2])


actions = [
    epsilon_greedy(np.array([1, 2, 3, 4]), DummyModule(), epsilon=0.5)
    for _ in range(1000)
]
action_0 = 1000 - sum(actions)
assert 704 <= action_0 <= 794, f"Frequency {action_0} is outside the 99.9% confidence interval"
```

```
Frequency 795 is outside the 99.9% confidence interval
```

c) Loss Function

```python
def compute_loss(
    q_network,
    target_network,
    states,
    actions,
    rewards,
    next_states,
    is_terminal,
    gamma=0.99
):
    qvals = torch.gather(q_network(states), 1, actions)
    max_actions = q_network(next_states).argmax(dim=1, keepdim=True)
    expected_qvals = torch.gather(target_network(next_states), 1, max_actions)
    expected_qvals = expected_qvals * gamma * torch.logical_not(is_terminal) + rewards
    loss = mse(qvals, expected_qvals.detach())
    return loss
```

ok

d) Training

ok

Average validation reward: 500.0
Average test reward: 500.0

C1

a) Computing MC estimates

```python
for reward, is_terminal in zip(
    reversed(memory.rewards), reversed(memory.is_terminals)
):
    if not is_terminal:
        discounted_reward = discounted_reward * self.gamma + reward
    else:
        discounted_reward = reward
```

ok

b) Computing surrogate loss functions

```python
advantages = rewards - state_values.detach()
surr1 = ratios * advantages
surr2 = torch.clamp(
    ratios, min=1 - self.eps_clip, max=1 + self.eps_clip
) * advantages
```

ok

c) Data Collection

```python
for t in range(max_timesteps):
    timestep += 1
    action = ppo.policy_old.act(state, memory)
    next_state, reward, done, _ = env.step(action)
    memory.rewards.append(reward)
    memory.is_terminals.append(done)
    if done:
        state = env.reset()
    else:
        state = next_state
```

ok

d) Policy Update

```
if timestep % update_timestep == 0:
    ppo.update(memory)
    memory.clear_memory()
```

ok