cse-584-HW2

Jiamu Bai

October 27, 2024

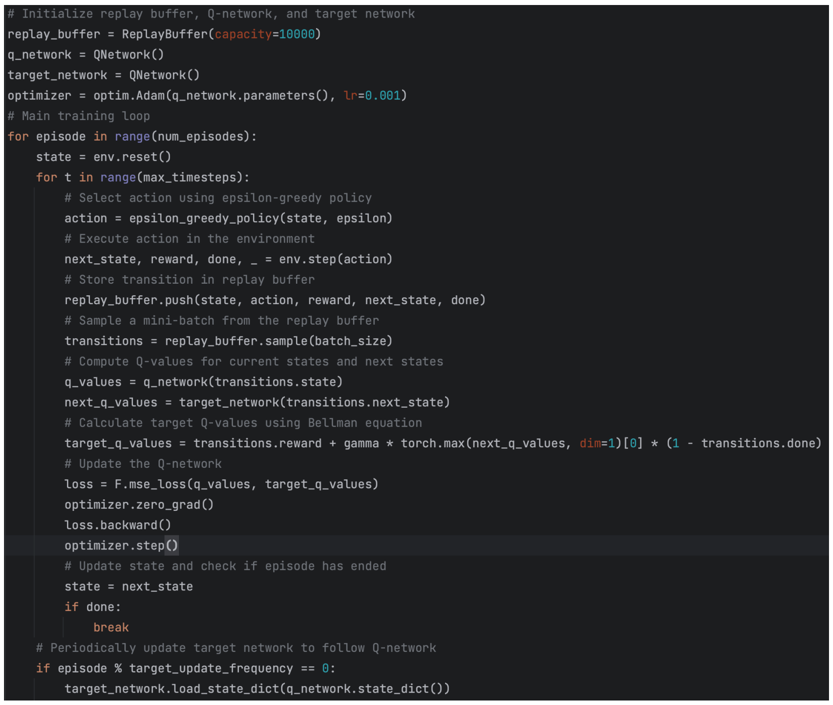
1 Abstract Overview

The DQN implementation trains an agent to navigate and maximize rewards within an environment (e.g., CartPole in OpenAI Gym). DQN uses a neural network to approximate the Q-value function, which evaluates potential actions. The training loop iteratively gathers experiences, updates the Q-network with these experiences, and applies exploration to balance exploration and exploitation. This model targets a stable, optimal policy by minimizing the difference between predicted Q-values and target Q-values.

2 DQN Code with Commentary

Here is a snippet of code highlighting the primary logic of DQN. Each line of code will be accompanied by explanations to clarify its function:

- Initialize Components: Replay buffer stores experience tuples; Q-network estimates Q-values; target network stabilizes training by periodically copying Q-network weights; optimizer adjusts weights to minimize Q-value error.
- Training Loop: Each episode, the agent observes the environment and selects actions to maximize cumulative reward.
- Action Selection: The epsilon-greedy policy balances exploration (random actions) and exploitation (best known actions).
- Experience Storage: After executing an action, the resulting experience (state, action, reward, next state, done flag) is saved.
- Mini-Batch Sampling: For efficient training, a mini-batch of experiences is sampled from the replay buffer.
- Q-Value Calculation: Q-values are calculated for the sampled batch, predicting the reward for each possible action.



```python
# Initialize replay buffer, Q-network, and target network
replay_buffer = ReplayBuffer(capacity=10000)
q_network = QNetwork()
target_network = QNetwork()
optimizer = optim.Adam(q_network.parameters(), lr=0.001)
# Main training loop
for episode in range(num_episodes):
    state = env.reset()
    for t in range(max_timesteps):
        # Select action using epsilon-greedy policy
        action = epsilon_greedy_policy(state, epsilon)
        # Execute action in the environment
        next_state, reward, done, _ = env.step(action)
        # Store transition in replay buffer
        replay_buffer.push(state, action, reward, next_state, done)
        # Sample a mini-batch from the replay buffer
        transitions = replay_buffer.sample(batch_size)
        # Compute Q-values for current states and next states
        q_values = q_network(transitions.state)
        next_q_values = target_network(transitions.next_state)
        # Calculate target Q-values using Bellman equation
        target_q_values = transitions.reward + gamma * torch.max(next_q_values, dim=1)[0] * (1 - transitions.done)
        # Update the Q-network
        loss = F.mse_loss(q_values, target_q_values)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Update state and check if episode has ended
        state = next_state
        if done:
            break
    # Periodically update target network to follow Q-network
    if episode % target_update_frequency == 0:
        target_network.load_state_dict(q_network.state_dict())
```

Figure 1: DQN Code with Commentary. The code is obtained from the following github link: https://github.com/vwxyzjn/cleanrl

- Target Q-Value Calculation: The Bellman equation computes the optimal Q-values by considering the reward from the current action plus discounted future rewards.
- Optimization Step: The Q-network is updated by minimizing the difference between the predicted Q-values and the target values using MSE loss.
- Update State and Check Completion: The environment state is updated for the next timestep, and if the episode ends, it resets.
- Target Network Update: Every few episodes, the target network is synchronized with the Q-network to maintain stability.