Double DQN Implementation Analysis

Debarshi Kundu

Abstract

The provided code implements a Double Deep Q-Network (Double DQN), an extension of the Q-learning algorithm. The primary objective of the Double DQN is to mitigate the overestimation bias observed in standard Q-learning by separating the action selection and evaluation processes during the target value computation. The algorithm is applied to solve reinforcement learning tasks, where an agent learns to make sequential decisions by maximizing cumulative rewards. The implementation leverages neural networks to approximate Q-values and utilizes mechanisms like experience replay and target networks for stability and efficiency during training. The code details the process of selecting actions, updating Q-values, and learning from sampled experience, demonstrating the effectiveness of Double DQN in producing more stable and accurate policies compared to traditional Q-learning approaches.

1 Introduction

Reinforcement Learning (RL) is a framework where agents learn optimal policies by interacting with an environment and maximizing a reward signal. Traditional Q-learning algorithms are known for their tendency to overestimate Q-values, which can lead to suboptimal policies. Double DQN addresses this issue by decoupling the action selection from evaluation, thus reducing overestimation bias. This report provides a high-level overview of the Double DQN implementation, including detailed comments on the core sections of the code.

2 Core Section Analysis

The following code snippet represents the key functions involved in the Double DQN process:

2.1 DQN Network Architecture

Listing 1: DQN Network

```
import torch
import torch.nn as nn

# Define the neural network architecture for the Q-value approximation.
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)  # First hidden layer with 64 neurons.
        self.fc2 = nn.Linear(64, 64)          # Second hidden layer with 64 neurons.
        self.out = nn.Linear(64, action_size) # Output layer that returns Q-values for each action.

    def forward(self, x):
        x = torch.relu(self.fc1(x))           # Apply ReLU activation function after the first layer.
        x = torch.relu(self.fc2(x))           # Apply ReLU activation function after the second layer.
        return self.out(x)                    # Output Q-values for the given state.
```

Explanation:

- The `DQN` class defines a neural network that approximates the Q-value function $Q(s, a)$.
- The architecture includes two hidden layers with ReLU activation functions, which introduce non-linearity to capture complex patterns in the data.
- The final layer outputs Q-values for each possible action, given an input state.

2.2 Experience Replay

Listing 2: Experience Replay

```
# Experience Replay buffer to store and sample experiences.
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity
        self.position = 0
    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

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

Explanation:

- The `ReplayBuffer` class stores past experiences, allowing the agent to sample batches for training.
- This process helps in breaking the temporal correlation between consecutive experiences, making the training more stable.
- Each experience consists of a state, action, reward, next state, and a boolean indicating if the episode ended.

2.3 Loss Calculation for Double DQN

Listing 3: Double DQN Loss Calculation

```
# Calculate the loss for the Double DQN algorithm.
def compute_loss(batch, model, target_model, gamma):
    states, actions, rewards, next_states, dones = batch
    states = torch.tensor(states)
    actions = torch.tensor(actions)
    rewards = torch.tensor(rewards)
    next_states = torch.tensor(next_states)
    dones = torch.tensor(dones, dtype=torch.bool)

    # Compute Q values for the current states using the online network.
    q_values = model(states)
    q_value = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

    # Compute the target Q values using the target network.
    with torch.no_grad():
        # Select the action with the highest value using the online network.
        next_q_values = model(next_states)
        next_actions = next_q_values.argmax(1)

        # Evaluate the selected action using the target network.
        next_q_value = target_model(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
        expected_q_value = rewards + gamma * next_q_value * (1 - dones)

    # Compute the loss between the estimated and target Q values.
    loss = nn.functional.mse_loss(q_value, expected_q_value)
    return loss
```

Explanation:

- This function computes the loss for training the Double DQN.
- `q_value`: Represents the Q-values of the current states and selected actions.
- `expected_q_value`: Represents the target value computed using the Double DQN approach, where action selection is done using the online model and evaluation is done using the target model.
- The loss is calculated using mean squared error (MSE) between the estimated and target Q-values, driving the network to learn more accurate value estimates.

3 Summary

This report examined a Double DQN implementation aimed at addressing overestimation biases present in standard Q-learning. The analysis provided a breakdown of the code's core components, such as the neural network architecture, experience replay mechanism, and the loss computation using Double Q-learning principles. Through this approach, the code achieves more stable learning by decoupling action selection from value evaluation, thereby producing better policies in reinforcement learning tasks.