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 Overview of the Problem

The core problem addressed by both the DQN (Deep Q-Network) and Double DQN algorithms involves learning optimal control policies for agents interacting with complex environments, such as Atari games. In such reinforcement learning (RL) tasks, the goal is for an agent to maximize its cumulative reward by selecting actions based on observed states.

Reinforcement learning involves learning from interactions with an environment to determine the best actions to take in each state to maximize future rewards. This setup can be formalized as a Markov Decision Process (MDP), where the agent's actions impact the state transitions and rewards. A key challenge in RL is learning effective policies directly from high-dimensional inputs (like raw pixel data from games) without any game-specific knowledge

3 Deep Q-Networks (DQN)

The DQN algorithm represents a significant advancement in the application of deep learning to reinforcement learning, allowing agents to learn directly from raw sensory inputs using a deep neural network. This network estimates the Q-value function $Q(s, a)$, which represents the expected future rewards for taking action a in state s. The key elements of DQN include:

- **Experience Replay**: Stores past interactions (state, action, reward, next state) in a replay buffer, and samples mini-batches of experiences during training. This helps break the correlation between consecutive observations, stabilizing the learning process.
- **Target Network**: Uses a separate target network to calculate target Q-values, which are updated periodically. This reduces the instability in learning by preventing rapid changes in the Q-value estimates[1, 2].

The DQN loss function is defined as:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right]$$

where θ represents the parameters of the online Q-network, θ^- are the parameters of the target network, γ is the discount factor, and D is the experience replay buffer.

In the provided code, the implementation of the experience replay buffer and target network is crucial for stability:

Listing 1: Double DQN Loss Calculation

```
# Experience Replay buffer to store and sample experiences.
class ReplayBuffer:
    ...
# Target Network Update (Periodically copy weights)
if timestep % TARGET_UPDATE == 0:
    target_model.load_state_dict(model.state_dict())
```

These components align with the mechanisms outlined in the DQN paper, ensuring stability in learning[1].

4 Challenges with DQN: Overestimation Bias

A major issue with DQN is the overestimation bias that arises during the Q-value updates. This bias occurs because the max operator in the DQN update simultaneously selects and evaluates actions. Since Q-values can be inaccurate during learning, the max operation tends to favor overestimated values, leading to overly optimistic value estimates. This can result in unstable training and suboptimal policies[2].

5 Double DQN: Addressing Overestimation Bias

The Double DQN algorithm builds upon DQN by decoupling the action selection from action evaluation in the target value calculation, thereby reducing overestimation. This is achieved by using the online network for action selection and the target network for action evaluation:

$$Y^{\text{DoubleDQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)$$

In this update rule:

- The action selection (arg max) is performed using the online network θ.
- The evaluation of the selected action is performed using the target network θ^-.

By separating these two steps, Double DQN reduces the likelihood of overestimating the action values, leading to more stable and accurate value estimates[2].

5.1 Key Sections in the Code

The Double DQN's advantage is clearly visible in the `compute_loss` function, where the target value computation differs from standard DQN:

Listing 2: Double DQN Loss Calculation

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

Here, `next_actions` are selected using the online network, and `next_q_value` is computed using the target network, implementing the decoupled selection and evaluation step of Double DQN[2].

6 How RL Solves the Problem

In both DQN and Double DQN, the agent learns to approximate the Q-values for different state-action pairs through interactions with the environment:

1. The agent observes a state s_t.
2. It selects an action a_t using an exploration-exploitation strategy like ϵ-greedy.
3. The environment responds with a reward r_{t+1} and a new state s_{t+1}.
4. This transition is stored in the replay buffer and used to update the Q-values through backpropagation.
5. In Double DQN, this update uses the decoupled selection and evaluation process, leading to more accurate Q-value estimation.

This iterative process enables the agent to learn policies that maximize cumulative rewards over time. The improvements introduced by Double DQN allow for a more precise learning of the optimal actions, particularly in complex environments like Atari games, where overestimation can significantly impact performance.

7 Core Section Analysis

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

7.1 DQN Network Architecture

Listing 3: 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.

7.2 Experience Replay

Listing 4: 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.

7.3 Loss Calculation for Double DQN

Listing 5: 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.

8 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.

References

[1] Jonathan Chung. Playing atari with deep reinforcement learning. *Comput. Ence*, 21:351–362, 2013.

[2] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.