HW2 : Analysis of Double DQN Implementation

Debarshi Kundu

Abstract

This report analyzes a Double Deep Q-Network (Double DQN)[3] implementation[1], an advanced reinforcement learning algorithm designed to address the overestimation bias in traditional Q-learning. The provided code applies this algorithm to solve decision-making problems, where an agent learns optimal policies through interaction with its environment. The Double DQN enhances stability and accuracy in learning by decoupling the action selection from the evaluation during target value computation. Key elements such as experience replay, target networks, and Q-value updates are dissected to highlight how the code effectively implements the algorithm. This report also includes a detailed explanation of the training process and critical sections of the code, illustrating a clear understanding of the Double DQN mechanism.

1 Introduction

Reinforcement Learning (RL) enables agents to learn optimal strategies by interacting with an environment and maximizing cumulative rewards. Q-learning is a widely used RL algorithm, but it is prone to overestimation errors when using function approximation, leading to suboptimal decisions. To address this, Double DQN introduces a modification by decoupling the action selection and evaluation steps, thus reducing overestimation bias. This report explores the Double DQN implementation, providing an in-depth analysis of the problem, the solution offered by Double DQN, and the key components of the provided code.

2 Problem Overview

In RL, the agent interacts with an environment modeled as a Markov Decision Process (MDP), where states, actions, rewards, and transitions determine the agent's experience. The goal is to learn a policy that maximizes the expected cumulative reward over time. The Q-learning approach estimates the action-value function $Q(s, a)$, which predicts the expected reward for taking action a in state s. However, Q-learning often overestimates Q-values due to the use of the max operator when computing target values. This overestimation can hinder learning and lead to unstable training, particularly when using deep neural networks.

3 Deep Q-Networks (DQN)

Deep Q-Networks (DQN) [2] combine Q-learning with deep neural networks to estimate Q-values from high-dimensional inputs, such as game frames. Key components of DQN include:

- **Experience Replay**: Stores agent experiences (state, action, reward, next state) in a buffer. During training, mini-batches are sampled from this buffer to break correlations between consecutive experiences, leading to more stable learning.
- **Target Network**: A separate network that periodically updates its weights to match those of the online network. This helps to stabilize Q-value targets during training.

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

4 Challenges with DQN: Overestimation Bias

The max operator in DQN can lead to overestimated Q-values, as it uses the same values to both select and evaluate actions. This bias can result in overly optimistic estimates, affecting the learning of optimal policies. The problem is particularly pronounced when training with high-dimensional inputs like raw pixel data from games, where slight inaccuracies can compound over time, leading to unstable training behavior.

5 Double DQN: Mitigating Overestimation Bias

Double DQN modifies the target calculation to separate the action selection from the evaluation:

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

This adjustment uses the online network θ to select the best action but evaluates this action using the target network θ^-. By decoupling these processes, Double DQN reduces the overestimation bias, leading to more accurate Q-value estimation and more stable training.

5.1 Key Sections in the Code

The core advantage of Double DQN is implemented in the `compute_loss` function:

Listing 1: 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)
```

In this snippet:

- `next_actions` are chosen using the online network.
- `next_q_value` is calculated using the target network for the selected action, implementing the Double DQN decoupling.

6 Core Components of the Code

6.1 Network Architecture

Listing 2: DQN Network Architecture

```
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 estimates the Q-value function $Q(s, a)$.
- It consists of two fully connected layers with ReLU activation to introduce non-linearity.
- The output layer produces Q-values for all possible actions in a given state.

6.2 Experience Replay Mechanism

Listing 3: 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, enabling the agent to sample mini-batches for training.
- This reduces the correlation between consecutive experiences and ensures that learning is more stable.
- The buffer stores transitions up to a fixed capacity, and new experiences overwrite older ones once the capacity is full.

6.3 Training Process Analysis

The training process of the Double DQN algorithm in the provided code involves several stages, each critical for the stability and performance of the learning process. Below is a detailed explanation of each stage along with relevant code snippets:

- **Initialization**: The training starts with the initialization of the online and target Q-networks, as well as the replay buffer, which stores past experiences for experience replay.

  Listing 4: Initialization of Networks and Replay Buffer

  ```
  # Initialize the Q-networks and the replay buffer.
  online_model = DQN(state_size, action_size)
  target_model = DQN(state_size, action_size)
  target_model.load_state_dict(online_model.state_dict())  # Synchronize target network with online
  target_model.eval()  # Set the target model to evaluation mode.

  replay_buffer = ReplayBuffer(capacity=10000)  # Replay buffer with specified capacity.
  ```

  The online model is used for action selection and learning, while the target model provides stable target values. The target model is updated less frequently to prevent rapid changes in target values.

- **Action Selection**: Actions are chosen using an ϵ-greedy policy, balancing exploration and exploitation. The agent selects a random action with probability ϵ and the action with the highest Q-value with probability $1 - \epsilon$.

  Listing 5: Action Selection using ϵ-greedy Policy

  ```
  def select_action(state, epsilon):
      if random.random() < epsilon:
          return random.randint(0, action_size - 1)  # Random action (exploration)
      else:
          with torch.no_grad():
              return online_model(state).argmax().item()  # Best action (exploitation)
  ```

  This approach ensures that the agent explores different actions at the beginning of training and gradually focuses more on exploiting the learned policy as training progresses.

- **Experience Storage**: After each action, the agent stores the transition (state, action, reward, next state, done) in the replay buffer for later use during training.

  Listing 6: Storing Transitions in Replay Buffer

  ```
  # Store the transition in the replay buffer.
  replay_buffer.push(state, action, reward, next_state, done)
  ```

  This storage allows the agent to sample random mini-batches of experiences, breaking the correlation between consecutive transitions and making the learning process more stable.

- **Batch Sampling and Learning**: A mini-batch of transitions is sampled from the replay buffer, and the `compute_loss` function calculates the loss between the current Q-values and the Double DQN target values.

  Listing 7: Sampling Batch and Computing Loss

  ```
  # Sample a mini-batch from the replay buffer.
  batch = replay_buffer.sample(batch_size)
  loss = compute_loss(batch, online_model, target_model, gamma)
  ```

  This step ensures that the agent learns from a diverse set of past experiences, improving the generalization of the learned policy.

- **Gradient Descent**: The loss computed is used to update the online network's weights through backpropagation using a gradient descent optimizer.

  Listing 8: Gradient Descent Step

  ```
  # Perform a gradient descent step.
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
  ```

  This step adjusts the parameters of the online network to minimize the difference between predicted and target Q-values, refining the agent's understanding of the environment.

- **Target Network Update**: To ensure stable learning, the target network's weights are periodically updated to match those of the online network.

  Listing 9: Target Network Update

  ```
  # Update the target network every few steps.
  if timestep % TARGET_UPDATE_FREQUENCY == 0:
      target_model.load_state_dict(online_model.state_dict())
  ```

This delayed update prevents the target values from changing too rapidly, contributing to a more stable convergence during training.

The training process combines these steps in a loop that iteratively refines the agent's policy. By leveraging experience replay and a stable target network, the Double DQN algorithm effectively mitigates the challenges of overestimation bias and instability in deep reinforcement learning.

7 Conclusion

This report provided a detailed analysis of a Double DQN implementation, focusing on the core problem of overestimation bias in Q-learning and how Double DQN addresses this issue. Through a breakdown of key code components like the neural network, experience replay, and the loss calculation process, we demonstrated how the Double DQN algorithm achieves more accurate value estimates and improves policy stability in reinforcement learning tasks.

References

[1] Denny Britz. Reinforcement learning implementations, 2017. GitHub repository.

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

[3] 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.