Deep Q Network (DQN) - RL

Abstract
This report details the implementation of a reinforcement learning (RL) algorithm called Deep Q Network (DQN). The DQN uses a neural network to approximate the Q-value function for a given state and action, enabling the agent to learn which actions to take in different states to maximize the cumulative reward.

The Deep Q Network (DQN) implementation from the TensorFlow-Reinforce repository provides a neural network-based Q-learning approach for solving reinforcement learning problems. The code is designed to work with OpenAI Gym environments and implements both standard DQN and Double DQN variants. This implementation uses TensorFlow 1.0 and follows the architecture proposed by Mnih et al. The key functionalities include experience replay for improved sample efficiency and target networks for stable learning, which are essential components of modern deep reinforcement learning systems. This report provides a high-level overview of the DQN code, followed by a detailed line-by-line analysis of the core sections.

The key functionalities of this implementation include experience replay, which enhances sample efficiency by allowing the agent to learn from past experiences multiple times, and target networks, which improve the stability of the learning process by providing more consistent target values. Additionally, the implementation utilizes epsilon-greedy exploration to ensure a balance between exploring new actions and exploiting known information.

Core Implementation Analysis

Below is the core section of the DQN implementation, specifically focusing on the main reinforcement learning components. The comments provide an in-depth explanation of each line, demonstrating the detailed working of the algorithm:

NeuralQLearner Class

```
# This class implements the Deep Q-Learning agent with experience replay and target networks.
class NeuralQLearner(object):
    def __init__(self, session,
                 optimizer,
                 q_network,
                 state_dim,
                 num_actions,
                 batch_size=32,
                 init_exp=0.5,  # initial exploration prob
                 final_exp=0.1,  # final exploration prob
                 anneal_steps=10000,  # N steps for annealing exploration
```

```
                 replay_buffer_size=10000):
        # Initializes the agent with the given environment, Q-network, and exploration
parameters.

        # Store all input parameters
        self.session = session
        self.optimizer = optimizer
        self.state_dim = state_dim
        self.num_actions = num_actions
        self.batch_size = batch_size
        self.init_exp = init_exp
        self.final_exp = final_exp
        self.anneal_steps = anneal_steps

        # Create experience replay buffer
        self.replay_buffer = deque(maxlen=replay_buffer_size)  # Replay buffer is used to
store past experiences, allowing the agent to learn from them multiple times, which helps in
stabilizing training by breaking temporal correlations between consecutive experiences.

        # Create Q networks
        self.q_network = q_network
        self.target_network = copy.deepcopy(q_network)  # The target network is a copy of the
Q-network, used to provide stable targets during training and improve convergence.

    def storeExperience(self, state, action, reward, next_state, done):
        # Store the transition in the replay buffer to allow the agent to replay past
experiences during training, which helps improve learning stability and efficiency.
        # Store transition in replay buffer
        self.replay_buffer.append((state, action, reward, next_state, done))

    # The `updateModel` function updates the Q-network using a mini-batch of experiences from
the replay buffer to improve learning stability.
    def updateModel(self):
        # Sample random minibatch from replay buffer
        minibatch = random.sample(self.replay_buffer, self.batch_size)

        # Unpack minibatch
        states = np.array([data[0] for data in minibatch])
        actions = np.array([data[1] for data in minibatch])
        rewards = np.array([data[2] for data in minibatch])
        next_states = np.array([data[3] for data in minibatch])
        dones = np.array([data[4] for data in minibatch])
```

```
        # Calculate target Q values
        target_q_values = self.target_network.predict(next_states)
        target_q_values[dones] = 0
        targets = rewards + self.gamma * np.max(target_q_values, axis=1)

        # Update Q network
        self.q_network.train(states, actions, targets)

        # Periodically update target network
        if self.train_iteration % self.target_update_freq == 0:
            self.target_network.copy_from(self.q_network)

    def getAction(self, state, epsilon):
        # This function uses epsilon-greedy action selection to balance exploration and
exploitation.
        # Epsilon-greedy action selection
        if random.random() < epsilon:
            return random.randint(0, self.num_actions - 1)
        else:
            q_values = self.q_network.predict([state])[0]
            return np.argmax(q_values)
```

This implementation demonstrates the important features of the DQN agent, such as the setup of neural networks, experience replay, and the main training loop.

DQN Agent Implementation

```
import sys
import numpy as np
from models import net
from utils import linear_schedule, select_actions, reward_recorder
from rl_utils.experience_replay.experience_replay import replay_buffer
import torch
from datetime import datetime
import os
import copy

# Define the DQN agent
class dqn_agent:
    def __init__(self, env, args):
        # Store the environment and hyperparameters
```

```
        self.env = env
        self.args = args

        # Define the neural network for the Q-values
        self.net = net(self.env.action_space.n, self.args)
        self.target_net = copy.deepcopy(self.net)

        # Create the experience replay buffer
        self.replay_buffer = replay_buffer(self.args.buffer_size)

        # Set initial exploration rate and schedule
        self.exploration_rate = self.args.init_exp
        self.exploration_rate_schedule = linear_schedule(self.args.init_exp,
self.args.final_exp, self.args.anneal_steps)

        # Optimizer setup
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=self.args.lr)

    def store_experience(self, state, action, reward, next_state, done):
        # Store the transition in the replay buffer
        self.replay_buffer.add(state, action, reward, next_state, done)

    def update_model(self):
        if len(self.replay_buffer) < self.args.batch_size:
            return

        # Sample a minibatch from the replay buffer
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.args.batch_size)

        # Convert to tensors
        states = torch.FloatTensor(states)
        actions = torch.LongTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)
        dones = torch.FloatTensor(dones)

        # Calculate the target Q values using the target network
        next_q_values = self.target_net(next_states).max(1)[0]
        next_q_values = next_q_values * (1 - dones)
        target_q_values = rewards + (self.args.gamma * next_q_values)

        # Calculate the current Q values from the main network
```

```python
        q_values = self.net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

        # Compute loss and update the main network
        loss = torch.nn.functional.mse_loss(q_values, target_q_values.detach())
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()



    def get_action(self, state):
        # Epsilon-greedy action selection
        if np.random.rand() < self.exploration_rate:
            return self.env.action_space.sample()
        else:
            state = torch.FloatTensor(state).unsqueeze(0)
            q_values = self.net(state)
            return q_values.max(1)[1].item()

    def update_target_network(self):
        # Update the target network with the main network's weights
        self.target_net.load_state_dict(self.net.state_dict())
```

Initialization Function

- The `__init__` function initializes the DQN agent with key components, such as the environment, neural network models (`net` and `target_net`), and the experience replay buffer.
- The exploration rate is set up with a schedule that gradually decreases from the initial exploration (`init_exp`) to the final exploration rate (`final_exp`) over a defined number of steps.
- The optimizer (`Adam`) is used to train the Q-network.

Experience Storage

- The `store_experience` method is used to add transitions (state, action, reward, next_state, done) to the experience replay buffer.

Model Update

- The `update_model` function samples a minibatch from the replay buffer and computes target Q-values using the target network.
- The target values are calculated by taking the maximum Q-value for the next state and adjusting for terminal states (`done`).
- The Q-values for the current state are calculated using the main Q-network, and the loss is computed as the Mean Squared Error (MSE) between the predicted and target Q-values.
- The loss is backpropagated to update the weights of the Q-network.

Action Selection

- The `get_action` method follows an epsilon-greedy strategy to select actions, balancing exploration and exploitation.
- If a random value is less than the current exploration rate, a random action is chosen; otherwise, the action with the highest predicted Q-value is selected.

Target Network Update

- The `update_target_network` method copies the weights from the main Q-network to the target network to periodically update it, providing stability during training.

Conclusion

The provided Deep Q Network implementation follows the fundamental principles of reinforcement learning as proposed by Mnih et al. Key features like experience replay and target networks are used to improve the stability and sample efficiency of the learning process. The epsilon-greedy action selection helps maintain a balance between exploration and exploitation, allowing the agent to find an optimal policy over time. This analysis captures the essential components and their role in the DQN algorithm, providing a foundational understanding of its implementation.

References

1. https://github.com/yukezhu/tensorflow-reinforce