# Understanding Reinforcement Learning Code

Xin Gu

The reinforcement learning code is chosen from PyTorch Reinforcement Learning (DQN) Tutorial[1].

## 1 High-level Overview

This code implements a Deep Q-Network (DQN) to train an agent on the CartPole-v1 environment using PyTorch. The DQN algorithm enables the agent to learn an optimal policy by 1) predicting expected rewards for each action and adjusting its predictions through experience, 2) a replay memory stores past experiences, allowing the agent to sample them for training, which stabilizes the learning process, 3) the Q-network predicts action values, while a target network provides stable estimates during updates. Through repeated interaction with the environment, the agent learns to keep the pole balanced by improving its policy iteratively, leading to longer balanced episodes over time.

## 2 Core Section Comments

```python
# Function to optimize the Q-network based on a batch of past experiences
def optimize_model():
    # Ensure there's enough data in replay memory for a full batch
    if len(memory) < BATCH_SIZE:
        return  # Exit if not enough transitions are stored

    # Sample a batch of transitions from replay memory
    transitions = memory.sample(BATCH_SIZE)
    # Reshape transitions
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states (next_state is not None)
    # Allows the model to ignore terminal states in the Q-value calculation
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None,
                                            batch.next_state)), device=device, dtype=torch.bool)
    # Concatenate all non-final next states into a single tensor
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])

    # Concatenate each component of the batch into tensors for batch processing
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
```

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#reinforcement-learning-dqn-tutorial

```python

    # Compute Q(s_t, a) - the Q-values for the actions taken in each state in the batch
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # Compute V(s_{t+1}) for all next states using the target network
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        # For non-final states, choose the action with the max Q-value in the target network
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1).values

    # Compute the expected Q values (target values) using the Bellman equation
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Compute the loss between the Q-values from policy_net and the target Q-values
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Perform a backward pass to compute gradients and update weights
    optimizer.zero_grad()  # Clear any previous gradients
    loss.backward()  # Backpropagate the loss to compute gradients

    # Clip gradients to a max value (100) to prevent exploding gradients
    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()  # Update policy_net's parameters based on gradients
```

Listing 1: One step of optimization

```python
# Main loop for training across multiple episodes
for i_episode in range(num_episodes):
    # Reset the environment at the beginning of each episode, getting initial state
    state, info = env.reset()
    # Convert state to a tensor for compatibility with PyTorch
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze(0)

    for t in count():
        # Select an action based on the current policy and epsilon-greedy exploration
        action = select_action(state)

        # Take the selected action and observe the next state, reward, and done flag
        observation, reward, terminated, truncated, _ = env.step(action.item())
        # Convert the reward to a tensor for consistency in optimization step
        reward = torch.tensor([reward], device=device)

        # Check if the episode is done (pole fell or cart went out of bounds)
        done = terminated or truncated

        # Set the next state; if the episode is done, next_state is None
        if terminated:
            next_state = None
        else:
            # Convert next state to tensor
```

```python
            next_state = torch.tensor(observation, dtype=torch.float32, device=device).unsqueeze(0)

        # Store the current transition (state, action, next_state, reward) in replay memory
        memory.push(state, action, next_state, reward)

        # Move to the next state for the next time step
        state = next_state

        # Perform one optimization step to improve the policy network based on sampled transitions
        optimize_model()

        # Soft update of the target network's weights for stability in training
        # target_net weights = tau * policy_net weights + (1 - tau) * target_net weights
        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key]*TAU + target_net_state_dict[key]*(1-TAU)
        target_net.load_state_dict(target_net_state_dict)

        # If episode is done, record its duration and break the loop
        if done:
            episode_durations.append(t + 1)
            plot_durations()  # Update the plot to show episode durations
            break
```

Listing 2: Training loop

# 3 Conclusion

In conclusion, the Rodinia "openmp/hotspot" benchmark showed significant sensitivity to CPU frequency changes, with a signifcant reduction in execution time as CPU frequency increased. Memory technology also impacted performance, but to a less degree. By analyzing the code, we know that Rodinia "openmp/hotspot" benchmark has a structured data access pattern, meaning good spatial and temporal locality, thus allowing the application to efficiently use the cache, resulting in lower cache miss rates.