DQNAgent Class Documentation

Overview

The `DQNAgent` class implements a Deep Q-Network (DQN) agent for reinforcement learning, inspired by the original DQN algorithm from Mnih et al. (2015) and adapted for dynamic pricing scenarios as referenced in Kastius & Schlosser (2022). Key features include:

- Use of continuous state representations (e.g., actual prices rather than discrete indices).
- Double DQN mechanism with a separate target network to reduce overestimation bias.
- Experience replay buffer for stable training.
- Huber loss (smooth L1) and gradient clipping for improved stability in noisy environments.
- Epsilon-greedy exploration with decay.

This implementation prioritizes stability and is suitable for multi-agent competitive settings, such as duopoly pricing simulations. It deviates from the exact rlpricing-master architecture (e.g., no dueling network, continuous states) to accommodate different environments, as noted in project guidelines (e.g., 2D states required).

Note: Hyperparameters can be adjusted if differing from reference implementations (e.g., MSE vs. Huber loss; choose based on environment needs).

Initialization

```python
def __init__(self, agent_id, state_dim=2, action_dim=15, seed=None):
```

Parameters

- `agent_id` (int): Agent identifier (e.g., 0 or 1 for duopoly settings).
- `state_dim` (int, optional): Dimension of the state space (default: 2 for own and competitor prices).
- `action_dim` (int, optional): Number of discrete actions (e.g., price grid size; default: 15).
- `seed` (int or None, optional): Random seed for reproducibility across NumPy, PyTorch, and Python's random module.

Internal Hyperparameters

- `gamma` (float): Discount factor for future rewards (default: 0.99).
- `epsilon` (float): Initial exploration rate (default: 1.0).
- `epsilon_min` (float): Minimum exploration rate (default: 0.01).
- `epsilon_decay` (float): Decay multiplier per update (default: 0.995).
- `learning_rate` (float): Adam optimizer learning rate (default: 0.0001).

- `batch_size` (int): Minibatch size for replay (default: 128).
- `memory_size` (int): Replay buffer capacity (default: 50000).
- `target_update_freq` (int): Steps between target network updates (default: 500).

Initializes the online Q-network, target network, optimizer, and replay buffer. If a seed is provided, it ensures deterministic behavior.

Network Architecture

The agent uses a simple feedforward neural network for Q-value approximation:

```
def _build_network(self):
    return nn.Sequential(
        nn.Linear(self.state_dim, 128),
        nn.ReLU(),
        nn.Linear(128, 256),
        nn.ReLU(),
        nn.Linear(256, 128),
        nn.ReLU(),
        nn.Linear(128, self.action_dim)
    )
```

- Input: State vector (e.g., `[own_price, competitor_price]`).
- Output: Q-values for each action.
- Activation: ReLU throughout.
- No dueling streams (value/advantage separation) in this variant for simplicity.

Methods

select_action

```
def select_action(self, state, explore=True):
```

Selects an action using epsilon-greedy policy.

Parameters

- `state` (tuple or array-like): Current state (e.g., price tuple).
- `explore` (bool, optional): If True, enables epsilon-greedy exploration (default: True).

Returns

- int: Selected action index.

Description Converts state to tensor, computes Q-values using the online network. With probability epsilon, chooses a random action; otherwise, selects the argmax Q-value. If `explore=False`, always greedy.

remember

```
def remember(self, state, action, reward, next_state, done):
```

Stores a transition in the replay buffer.

Parameters

- `state` (tuple or array-like): Current state.
- `action` (int): Taken action.
- `reward` (float): Received reward.
- `next_state` (tuple or array-like): Next state.
- `done` (bool): Episode termination flag.

Description Appends the transition as a tuple to the deque buffer (FIFO with fixed size).

replay

```
def replay(self):
```

Performs a training step if buffer is sufficient.

Returns

- None or float: Returns None if buffer too small; otherwise, the computed loss.

Description Samples a batch, computes current and target Q-values using Double DQN logic: - Selects next actions via online network. - Evaluates them via target network. - Applies Bellman equation for targets. - Uses Huber loss and gradient clipping (max norm 1.0). - Updates online network; increments step counter and checks for target update.

update_target_network

```
def update_target_network(self):
```

Copies weights from online to target network.

Description Hard update to stabilize targets; called periodically based on `target_update_freq`.

update_epsilon

```python
def update_epsilon(self):
```

Decays the exploration rate.

Description Multiplies epsilon by decay factor, clamped to `epsilon_min`. Typically called per episode.

save

```python
def save(self, filepath):
```

Saves model checkpoint.

Parameters

- `filepath` (str): Path to save the .pth file.

Description Saves state dicts for networks and optimizer, plus epsilon and steps.

load

```python
def load(self, filepath):
```

Loads model checkpoint.

Parameters

- `filepath` (str): Path to load the .pth file.

Description Restores state dicts, epsilon, and steps for resuming training.

Usage Notes

- **Training Loop**: In simulations (e.g., duopoly), call `select_action`, `remember` after env step, `replay` per step, and `update_epsilon` per episode.
- **Multi-Agent**: Use agent_id for distinction; states are agent-specific (own vs. opponent).
- **Adaptations**: Continuous states suit pricing; for discrete indices (as in references), normalize accordingly. Huber loss chosen for stability over MSE.
- **Dependencies**: Requires PyTorch, NumPy. Seed for reproducibility.

For reference implementations, see rlpricing-master (ql.py) or adaptations from Deepseek/Grok discussions (e.g., MSE/Huber choice, no "ig" flags needed here).