Introduction

This documentation outlines the implementation of a Deep Q-Learning (DQN) agent in a 3D environment using Godot, aimed at simulating goal-oriented movement and reward-driven actions. The agent learns to navigate toward a target, or "power node," by receiving feedback based on its proximity to the target and adapting its behavior over time.

Understanding Reinforcement Learning (RL) and DQN

Reinforcement Learning is a foundational concept in machine learning, where agents learn by interacting with an environment, taking actions, and maximizing cumulative rewards. Q-Learning, specifically, is a model-free RL algorithm that finds the optimal action-selection policy for an agent by learning Q-values (expected rewards). DQN extends Q-Learning by using a neural network to approximate Q-values, handling complex, high-dimensional spaces.

Research and Application of DQN with NNET

To achieve this agent's behavior, I researched how DQNs operate, exploring epsilon-greedy policies, discount factors, experience replay, and target networks. By understanding these components, I structured the agent's learning process with two key networks (Q and target networks) using the NNET library in Godot, which supports efficient matrix operations and neural network structures essential for this implementation.

Code Structure and Implementation

Agent Setup and Initialization

The agent is built upon the RigidBody3D class in Godot, providing it with physical properties and behaviors. Key variables and structures include:

- **Action Control**: Controls agent motion in response to inputs.
- **Positioning and Time Variables**: rocket_position and current_time track location and elapsed time.
- **DQN Parameters**: epsilon (exploration), gamma (discount factor), replay_buffer (stores experience), and batch_size.

The agent initializes the Q-network and target network with 6 input nodes, a hidden layer of 64 nodes, and 4 output nodes, representing possible actions (move right, left, back, forward). Both networks use the Adam optimizer and ReLU activations, which will work for the current environment setup.

Action Selection with Epsilon-Greedy Policy

The agent chooses actions based on an epsilon-greedy strategy:

1. **Exploration (Random Action)**: With probability epsilon, the agent picks a random action, encouraging diverse experiences.
2. **Exploitation (Best Action)**: Otherwise, it selects the action with the highest Q-value predicted by the Q-network.

Epsilon decays over time, encouraging the agent to exploit learned behaviors as it gains more experience.

Reward Mechanism

The reward system is essential for guiding the agent's behavior:

- **Proximity Reward**: A significant reward is given when the agent reaches the target.
- **Step Penalty**: Small negative rewards discourage the agent from stalling.
- **Distance-based Reward**: The agent is rewarded based on distance reduction to the target. If the agent moves away, it incurs a penalty.

Experience Replay and Training Process

To stabilize learning, the agent uses an experience replay:

1. **Buffer Storage**: The agent stores each (state, action, reward, next_state, done) tuple in the replay buffer.
2. **Training Batch**: Periodically, a batch of experiences is drawn from the buffer to train the Q-network, reducing bias from sequential data.
3. **Q-Target Update**: Each experience is processed to compute the Q-target, which represents the expected Q-value. The target network is periodically synced with the Q-network to ensure stability.

Function Descriptions

1. **_ready()**: Sets up initial parameters and loads the neural networks. Copies weights from Q to target network to sync them initially.
2. **_physics_process(delta)**: Core function that triggers each frame. It updates position, computes actions, executes them, calculates rewards, and stores experiences.
3. **choose_action(current_state)**: Implements epsilon-greedy policy to either explore or exploit based on Q-values from the network.
4. **store_experience(state, action, reward, next_state, done)**: Appends experiences to the replay buffer, maintaining a buffer size limit.
5. **get_state()**: Collects the current state parameters, including position and velocity of the agent and the target.

6. **execute_action(action)**: Maps each action to a physical force applied to the agent.
7. **get_reward()**: Calculates the reward based on the agent's proximity and movement relative to the target.
8. **is_done()**: Checks for completion criteria, primarily used here to stop training if a maximum condition (like a bomb count) is reached.
9. **train_dqn()**: Core DQN training loop. Updates the Q-network using a mini-batch from the replay buffer, adjusting weights based on computed target Q-values.

Current Problems

- **Inconsistent Learning**: Agent sometimes veers off course. Likely needs tweaks to reward function, epsilon decay, or training frequency for better stability.
- **Replay Buffer Loss**: Older, possibly useful experiences get overwritten in the buffer. A prioritized replay system might preserve critical moments longer.
- **Target Network Sync**: Updating target network every so often may lead to outdated actions. More frequent syncing could make actions more relevant.
- **Exploration vs. Exploitation**: Epsilon decay may be too fast or too slow, affecting exploration.
- **Reward Jitter**: High penalties/rewards can make the agent jitter near the target. Fine-tuning reward values could smooth out behavior.
- **Performance in Godot**: Physics timing and DQN steps don’t always align perfectly, leading to delays. Code optimization may help this sync better with Godot’s physics engine.

Summary

This DQN agent in Godot, powered by NNET, reflects a structured approach to reinforcement learning where the agent improves through trial, error, and cumulative reward optimization. Key features like experience replay, epsilon decay, and target network stabilization enable it to learn and refine its decision-making process in a 3D environment, successfully moving toward defined goals.