Pseudocode for 3x3 Tic - Tac - Toe Using Deep Q - Networks

// --- System Components ---

// Environment (TicTacToeEnv)

```
CLASS TicTacToeEnv (inherits from gym.Env):
  BoardState: 3x3 grid (0: empty, 1: Agent/X, 2: Opponent/O)
  CurrentPlayer: 1 (Agent/X) initially
  ActionSpace: Discrete(9) (indices 0-8 for cells)
  ObservationSpace: Box(low=0, high=1, shape=(3, 9)) (flattened board layers for Player 1, Player 2,
Current Player)
  RuleBasedOpponentLogic: Function to choose opponent's move (win if possible, block if necessary,
else random)

  METHOD __init__():
    Initialize board empty
    Set current player to 1
    Define action and observation spaces

  METHOD reset(seed=None, options=None):
    Reset board to empty
    Set current player to 1
    Return initial observation and empty info dict

  METHOD step(action):
    // Agent's turn (CurrentPlayer = 1)
    IF action is illegal (out of range or cell not empty):
      Return current_obs, -1.0 (penalty), done=True, False, info={"Invalid move"}

    Place Agent's mark (1) on board at action
    Check if Agent wins:
      IF Agent wins:
        Return current_obs, 1.0 (win reward), done=True, False, info={"winner": 1}

    Check for Draw:
      IF Draw:
        Return current_obs, 0.5 (draw reward), done=True, False, info={"Draw"}

    // Opponent's turn (Opponent = 2)
```

```
    Get available moves
    IF available moves exist:
      Choose Opponent's action using RuleBasedOpponentLogic
      Place Opponent's mark (2) on board
      Check if Opponent wins:
        IF Opponent wins:
          Return current_obs, -1.0 (loss penalty), done=True, False, info={"winner": 2}

      Check for Draw:
        IF Draw:
          Return current_obs, 0.5 (draw reward), done=True, False, info={"Draw"}

    // If game not over
    Return next_obs, 0.0 (step penalty, if any, but 0 in this code), done=False, False, info={}

  METHOD _get_obs():
    Flatten board
    Create 3 layers: Player 1 positions, Player 2 positions, Is Current Player 1
    Stack layers to form observation (shape 3x9)

  METHOD check_win(player, board=None):
    Check rows, columns, diagonals for 3 consecutive marks of 'player'

  METHOD check_draw():
    Check if board is full and no winner

  METHOD get_available_moves():
    Find and return indices of empty cells

  METHOD rule_based_opponent():
    Implement opponent's move selection logic

  METHOD render():
    Print text-based board
```

// Agent (Stable Baselines3 DQN Model)

```
CLASS DQN_Agent:
  NeuralNetwork (Q-Network): Maps observation (3x9) to Q-values for 9 actions
  TargetNetwork: A copy of the Q-Network, updated less frequently
  ReplayBuffer: Stores past experiences (s, a, r, s', done)
  Hyperparameters: learning_rate, buffer_size, gamma, etc.
  ExplorationStrategy: Epsilon-Greedy (handled internally by SB3 during .learn)

  METHOD __init__(policy="MlpPolicy", env, ...):
```

```
    Initialize neural networks, replay buffer, hyperparameters
    Associate with environment

  METHOD learn(total_timesteps):
    // Training Loop (handled internally by SB3)
    Loop for total_timesteps:
      Get current observation from environment.
      Agent chooses action using Epsilon-Greedy (based on Q-Network)
      Send action to environment (env.step)
      Environment returns next_obs, reward, done, ...
      Store (obs, action, reward, next_obs, done) in ReplayBuffer
      Perform periodic updates:
        Sample batch from ReplayBuffer
        Calculate target Q-values using TargetNetwork:
          target_Q = reward + gamma * max(TargetNetwork(next_obs)) (if not done)
          target_Q = reward (if done)
        Calculate loss between current Q-values (from Q-Network) and target_Q
        Update Q-Network weights using optimizer (e.g., Adam)
      Update TargetNetwork weights periodically (copy from Q-Network)
      Decay epsilon (handled internally by SB3)
      IF episode done: Reset environment

  METHOD predict(observation, deterministic=False):
    // Action Selection for Inference/Evaluation
    IF deterministic is True (evaluation/human play):
      Pass observation through Q-Network
      Choose action with the highest Q-value
    ELSE (exploration during training):
      Use Epsilon-Greedy logic (random action with probability epsilon, else greedy) - *Note: The
.learn method uses this internally. You typically use predict with deterministic=True after training.*
    Return chosen action
```

// --- Main Execution Flow ---

// Setup
Create TicTacToeEnv instance
Wrap environment with Monitor (for logging, handled by SB3)
Create DQN_Agent model, associating it with the environment and setting hyperparameters

// Training
Call model.learn(total_timesteps) // Executes the internal SB3 training loop

// Save Trained Model

```
Call model.save("model_filename")
```

// Evaluation

```
Load the trained model (often evaluated using a separate function like evaluate_policy from SB3)
Call evaluate_policy(model, env, n_eval_episodes) // Runs games greedily, reports average reward
```

// Play Against Human

```
FUNCTION play_human_vs_agent():
  Create a new TicTacToeEnv instance for human play
  Load the trained DQN model
  Reset the environment
  Display board indices

  Loop WHILE game is NOT done:
    Render the board
    IF CurrentPlayer is Human (1):
      Get action input from human (1-9, with validation)
      IF human quits: Exit function
      Convert human input to action index (0-8)
    ELSE IF CurrentPlayer is Agent (2):
      Print "Agent is thinking..."
      Get action from the loaded DQN model using predict(deterministic=True)

    Send chosen action (human or agent) to env.step()
    Receive next_obs, reward, done, _, info
    Update current observation for agent's next turn

  Render final board
  Announce game result (Win, Loss, Draw)
```

// Entry point

```
IF script is run directly:
  Perform Setup, Training, Saving, Evaluation steps
  Call play_human_vs_agent()
```