Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Assignment 3
CS7IS2 Artificial Intelligence

Ujjayant Kadian (22330954)

April 5, 2025

Introduction

This report explores the implementation and evaluation of artificial intelligence agents in two well-known board games: Tic Tac Toe and Connect 4. The main focus is on comparing the performance of traditional search-based algorithms, such as Minimax (with and without alpha-beta pruning), against reinforcement learning methods, specifically Q-Learning.

Both games were developed from scratch using modular, object-oriented Python code, which allowed for a clear separation between the core game logic and the graphical interface. The integration of AI agents was followed by systematic experiments under various conditions, including matches against semi-intelligent default opponents and direct competitions between the AI agents.

The evaluation highlights significant trade-offs between learning-based and deterministic approaches, considering factors such as strategy depth, scalability, training overhead, and real-time performance. The findings provide insights into the suitability of each algorithm type for different problem scales and complexity levels, while also addressing specific implementation nuances and performance bottlenecks.

Game Implementations

General Setup

The games, Tic Tac Toe and Connect 4, were implemented in Python, leveraging `numpy` for efficient board state manipulation and `pygame` for rendering interactive graphical interfaces. Both games were implemented entirely from scratch rather than adapting open-source code, enabling complete control over game logic and integration with AI agents. This approach was justified by the necessity to clearly separate and customize the core mechanics, facilitating a clean integration with different algorithms (Minimax, Q-Learning, Semi-Intelligent).

Tic Tac Toe

Board Representation and Rules

The Tic Tac Toe board is represented as a 3x3 numpy array initialized with zeros. Player 1 (X) and Player 2 (O) are represented numerically as 1 and 2, respectively. The game adheres strictly to classic Tic Tac Toe rules, where players alternate placing their symbols aiming to align three horizontally, vertically, or diagonally.

Move Validation and Action Space

Valid moves are identified by inspecting unoccupied cells in the numpy array. The action space consists of tuples representing cell coordinates (row, col) for each empty space on the board.

Game Loop and Turn Handling

The game loop handles alternating player turns, updates the board state upon valid moves, and checks game termination conditions. It uses clearly defined methods such as `make_move(move)` for updating game states, `_check_game_over()` to verify end conditions, and `get_winner()` to identify the winner.

Connect 4

Board Representation and Gravity Mechanic

Connect 4 employs a 6x7 numpy array to represent the board, initialized with zeros. Each player's pieces, marked numerically as 1 or 2, "drop" to the lowest available space within a selected column, mimicking the gravity-based mechanics of the traditional Connect 4 game.

Move Validation and Win/Draw Logic

Moves are validated by checking column availability. The method `get_legal_moves()` identifies columns with open slots. The `_check_game_over()` method assesses the board after each move for vertical, horizontal, or diagonal connect-four conditions, or a draw scenario if no moves remain.

Game Loop Structure

The Connect 4 game loop sequentially manages player turns, processes moves using `make_move(col)`, and updates the game state. The loop includes logic for animation effects via the `_start_animation(col)` and `_update_animation()` methods, enhancing visual feedback during gameplay.

Reusability and Modularity

Separation of Game Logic and UI

A deliberate design decision was to separate game logic from the user interface (UI) by encapsulating core mechanics within dedicated classes (`TicTacToe` and `Connect4`) independent of graphical handling (`TicTacToeUI` and `Connect4UI`). This modular structure enables easy maintenance and allows the core logic to be reused or adapted for different interfaces or integrations without affecting the gameplay integrity.

Unified Player Types and Game Modes

Both games share common enumerations for player types (`HUMAN`, `AI`, `SEMI_INTELLIGENT`) and game modes (`HUMAN_VS_HUMAN`, `HUMAN_VS_AI`, `HUMAN_VS_SEMI`, `AI_VS_SEMI`, `AI_VS_AI`). The game modes define the interaction dynamics: `HUMAN_VS_HUMAN` allows two players to compete against each other, `HUMAN_VS_AI` pits a human player against an AI opponent, `HUMAN_VS_SEMI` features a human player against a semi-intelligent AI, `AI_VS_SEMI` involves a semi-intelligent AI competing against a fully intelligent AI, and `AI_VS_AI` allows two AI agents to play against each other. This design simplifies configuration management and provides consistent player behavior across both games, effectively supporting various experimental setups involving Minimax, Q-Learning, and Semi-Intelligent agents.

Interface and Interaction

The game classes provide clearly defined methods (`get_state()`, `get_legal_moves()`, `make_move()`, `is_game_over()`, `get_winner()`) that standardize interaction with AI agents. Agents can integrate seamlessly by using these methods to observe the game state, validate and execute moves, and assess game termination. The effectiveness of this design choice lies in its simplicity and clarity, which significantly ease the development and testing of different AI strategies while ensuring straightforward compatibility across both game environments.

Overall, these design decisions and implementations effectively supported modularity, facilitated diverse integration, and allowed focused analysis on algorithmic performance, crucial for the comparative assessment of Minimax and Q-Learning strategies explored in subsequent sections.

Algorithm Implementations

Overview

Two distinct algorithms were implemented and systematically compared: **Minimax** (with optional alpha-beta pruning) and **Q-Learning**. Minimax is a deterministic, exhaustive search algorithm suitable for finite, turn-based adversarial games, systematically exploring game states to make optimal decisions. In contrast, Q-Learning is a reinforcement learning approach, where optimal strategies emerge from repeated gameplay and feedback-based learning. This fundamental contrast: between exhaustive search and incremental learning enables meaningful comparative insights.

Minimax Algorithm

The core idea behind Minimax is to minimize the possible loss for a worst-case scenario. When it's the player's turn, they will choose the move that maximizes their minimum gain (hence "minimax"). Conversely, when it's the opponent's turn, they will choose the move that minimizes the player's maximum gain.

In this context, the game tree represents all possible moves in the game, where each node corresponds to a game state. The root node represents the current state of the game, and each child node represents a possible move leading to a new game state. The depth of the tree refers to how many moves ahead the algorithm evaluates; a greater depth allows for a more thorough analysis of potential future game states.

Pruning, specifically alpha-beta pruning, is an optimization technique for the Minimax algorithm that reduces the number of nodes evaluated in the game tree. It eliminates branches that cannot possibly influence the final decision, allowing the algorithm to run more efficiently by skipping unnecessary calculations.

Tic Tac Toe

Minimax for Tic Tac Toe was implemented in both alpha-beta pruning and non-pruning versions to facilitate experimentation. Given Tic Tac Toe's manageable state space, a complete exploration was feasible with a maximum depth of nine moves.
Evaluation Heuristic:

- **Terminal States:** +100 for wins, -100 for losses, 0 for draws.
- **Intermediate States:** Heuristic scores were assigned based on potential winning alignments, particularly rewarding configurations such as "two-in-a-row" opportunities.

Why This Heuristic Works: Tic Tac Toe's simplicity allows precise evaluations, where each potential alignment strongly influences the outcome. By emphasizing near-complete alignments, the Minimax can decisively exploit immediate threats or opportunities.

Connect 4

Due to Connect 4's complexity, the Minimax algorithm was implemented with adjustable maximum depths to maintain computational feasibility, alongside optional alpha-beta pruning to significantly reduce computational overhead.
Evaluation Heuristic:

- **Window-based evaluation:** Assesses four-cell windows horizontally, vertically, and diagonally, assigning points proportional to the number of pieces a player controls within each window.
- **Positional weighting:** Strongly favors central columns, aligning with known Connect 4 strategies for maximizing potential winning combinations.

Why This Heuristic Works: Connect 4 strategies commonly revolve around central control and creating multiple simultaneous threats. Evaluating windows effectively captures immediate threats and strategic potential, ensuring the Minimax selects optimal strategic moves (like in the case of tic tac toe).

Q-Learning Algorithm

As explained earlier, Q-Learning is a reinforcement learning algorithm that enables agents to learn optimal actions through trial and error by interacting with their environment. The algorithm utilizes a Q-table, which is a data structure that maps each possible game state to a set of action values, representing the expected utility of taking each action from that state. The Q-table effectively stores the rewards associated with each action taken in a given state, allowing agents to make informed decisions based on previously learned outcomes. Agents were thoroughly trained across thousands of episodes before actual evaluation against the default opponents or the minimax agent, as untrained Q-Learning agents default to random behavior due to uninitialized Q-tables.

Hyperparameters

Hyperparameters are crucial settings that influence the learning process of Q-Learning agents. They help balance the trade-offs between learning speed and the quality of the learned policy.
Learning Rate (α): This parameter determines how much new information overrides old information. A higher learning rate means the learns quickly from new experiences, while a lower rate allows for more gradual learning.

- Tic Tac Toe: 0.3, Connect 4: 0.2.

Discount Factor (γ): This factor represents the importance of future rewards. A value closer to 1 emphasizes future rewards, encouraging the to consider long-term benefits, while a value closer to 0 focuses on immediate rewards.

- Tic Tac Toe: 0.9, Connect 4: 0.95.

Epsilon (ϵ) (Exploration Rate): This parameter controls the exploration-exploitation trade-off. Exploration involves trying new actions to discover their effects, while exploitation involves choosing the best-known actions based on current knowledge. Starting at 0.3 and decaying to 0.01 allows the to initially explore a variety of actions and gradually focus on exploiting the best strategies as it learns.

Reward Structure

Designed to incentivize efficient and effective gameplay:

- **Win:** +1.0
- **Loss:** -1.0
- **Draw:**
  - Tic Tac Toe: +0.2 (as draws are relatively common and better than losses)
  - Connect 4: 0.0
- **Per-move penalty:**
  - Tic Tac Toe: -0.05 (to encourage rapid wins)
  - Connect 4: -0.01 (to slightly favor quicker resolutions)

Game-Specific Heuristics

Tic Tac Toe:

- **Symmetry Exploitation:** The leveraged board symmetries, greatly accelerating learning by applying knowledge gained from one state to equivalent symmetric states. This significantly reduced the Q-table size and improved learning efficiency. (Similar to the minimax agent)

Connect 4:

- **Immediate Threat Detection:** Identifies potential winning moves and opponent threats, allowing the to immediately capitalize or defend.
- **Double Threat Detection:** Prioritizes moves that simultaneously create multiple threats, increasing the strategic complexity of moves.
- **Center Control:** Explicitly rewards occupying the central column, a critical strategic advantage in Connect 4.

These heuristics are essentially the same as for Minimax agents. They dramatically enhanced Q-learning performance, aligning learned strategies closely with human-like strategic reasoning.

Default Opponent Agents

Semi-intelligent default opponents served as challenging baselines, superior to random agents due to their incorporation of fundamental game-specific strategies.
Tic Tac Toe Default Opponent:

- Immediately seizes winning opportunities or blocks imminent threats.
- Prefers central positions and corners to edges, reflecting basic strategic principles.

Connect 4 Default Opponent:

- Checks for immediate winning or blocking moves first.
- Prioritizes the central columns, recognizing their strategic value.
- Resorts to random moves only when no strategic advantage or threat is evident.

These default opponents provided more realistic benchmarks, enhancing the experimental validity of evaluating Minimax and Q-Learning agents.

Common Interfaces and Interaction

All algorithms used appropriate functions (`get_state()`, `get_legal_moves()`, `make_move()`) from the game classes (see the previous section), ensuring smooth and interchangeable agent-game interactions. The unified interaction workflow simplified the experimental setup:

1. Retrieve the current state via `get_state()`.
2. Obtain available actions through `get_legal_moves()`.
3. Select the best action using algorithm-specific logic.
4. Execute the move through `make_move()`.

This clear structure facilitated direct and fair comparisons between algorithms, ensuring efficient execution of experimental scripts and reliable interpretation of results.

Experimental Context and Evaluation

Comprehensive experimental scripts (one script to compare each algorithm against the default opponent and one script to compare each algorithm against each other) were used to systematically evaluate performance under various conditions:

- **Configurations:** Flexibly configured via command-line arguments, including selection between Minimax and Q-learning, choice of alpha-beta pruning, depth limits, training episodes, and exploration parameters.
- **Training and Evaluation Cycles:** Q-learning agents were methodically trained (if chosen) over numerous episodes with periodic evaluations, clearly demonstrating learned strategic competence.
- Timeout mechanisms were used, preventing excessively long computations, especially pertinent for Minimax for connect 4 with extensive search depths (explained in the next section).

Scalability Considerations for Connect 4

Connect 4 presents severe scalability challenges for both the Minimax and Q-Learning algorithms due to the vastness of its state and action spaces.

From a theoretical standpoint, Connect 4 is played on a 6×7 grid, totaling 42 cells. A complete game can last up to 42 moves, and on each turn, a player can choose among up to 7 columns (depending on which are still available). This gives a branching factor of up to 7 and leads to a state space complexity of:

$$O(c^r) = 7^{42} \approx 1.4 \times 10^{35}$$

In comparison, Tic Tac Toe has at most $9! = 362,880$ states, a tiny fraction of Connect 4's complexity. This makes exhaustive methods like full-depth Minimax or full Q-table coverage computationally infeasible.

In practice, when I attempted to run Minimax without depth limitation for Connect 4, the entered an endless evaluation loop, even with alpha-beta pruning enabled. Early game states, in particular, have the highest branching factor and result in an enormous game tree. To address this, I introduced the timeout configuration option as explained earlier.

The same scalability concerns affect Q-Learning. A tabular Q-Learning maintains a lookup table with entries for each (state, action) pair. While such an approach is tractable for Tic Tac Toe, it becomes unrealistic for Connect 4. Given the immense number of possible board configurations, the Q-table would require an exorbitant amount of memory and training episodes to fill meaningfully. Moreover, due to the sparse visitation of most states during training, the may fail to generalize well or converge efficiently. To make Q-Learning viable, I relied on heavy exploration during training (via ϵ-greedy action selection) and a more sophisticated heuristic for action selection during evaluation.

Evaluation

Metrics Used:

The evaluation framework employed a comprehensive set of metrics to assess both the performance and computational characteristics of the agents:

- **Game Outcome Metrics:**
  - Win/Loss/Draw rates to measure strategic effectiveness
  - Total games played to ensure statistical significance
  - Average moves per game to assess strategic efficiency
- **Computational Performance Metrics:**
  - Time per move (average, min, max) to evaluate real-time decision-making capability
  - Game duration to assess overall computational efficiency
  - States explored (for Minimax) to measure search space coverage
- **Q-Learning Specific Metrics:**
  - Episode rewards to track learning progress
  - Periodic evaluation results to measure improvement over time
  - Q-table memory usage to assess space complexity

These metrics were particularly suitable because they:

- Provide both strategic (win rates) and operational (time/space complexity) insights
- Enable direct comparisons between different types and configurations
- Capture the learning progression of Q-Learning agents
- Allow for scalability analysis through state exploration tracking
- Support real-world applicability assessment via timing measurements

The metrics were systematically collected using a dedicated `MetricsManager` class, ensuring consistent measurement and logging across all experiments. This approach enabled both detailed analysis of individual games and aggregate performance evaluation across multiple trials.

Experiment Results

The experiment scripts were run while calculating the above metrics and the following results were obtained:

Game	Match Type	Type	Player	Wins	Losses	Draws	Win Rate	Games Played	Avg. Moves	Time/ Move (s)	Game Dur. (s)	States Explored
Connect 4	agent_vs_default	Q-Learning - Trained	Player 1	75	22	3	0.7500	100	21.2500	0.0009	0.0200	–
Connect 4	agent_vs_default	Minimax - with Pruning (Depth 5)	Player 1	10	0	0	1.0000	10	22.4000	0.1635	3.6600	208161
Connect 4	agent_vs_default	Minimax - no Pruning (Depth 5)	Player 2	10	0	0	1.0000	10	22.0000	1.2572	27.6600	1357570
Tic Tac Toe	agent_vs_default	Q-Learning - Untrained	Player 1	1	9	0	0.1000	10	6.5000	0.0000	0.0000	–
Tic Tac Toe	agent_vs_default	Q-Learning - Trained	Player 2	0	0	100	0.0000	100	9.0000	0.0000	0.0000	–
Tic Tac Toe	agent_vs_default	Minimax - no Pruning (Depth 5)	Player 2	0	0	10	0.0000	10	9.0000	0.0165	0.1500	86940
Tic Tac Toe	agent_vs_default	Minimax - with Pruning (Depth 5)	Player 2	0	0	10	0.0000	10	9.0000	0.0017	0.0200	10331
Tic Tac Toe	agent_vs_default	Minimax - with Pruning (Depth 5)	Player 1	0	0	10	0.0000	10	9.0000	0.0065	0.0600	36494
Tic Tac Toe	agent_vs_default	Minimax - with Pruning	Player 2	0	0	10	0.0000	10	9.0000	0.0014	0.0100	25484
Tic Tac Toe	agent_vs_default	Minimax - no Pruning and no depth	Player 2	0	0	10	0.0000	10	9.0000	0.0272	0.2500	565620

Table 1: Performance of different variants against default opponents in Tic Tac Toe and Connect 4.

Algorithms vs Default Opponents

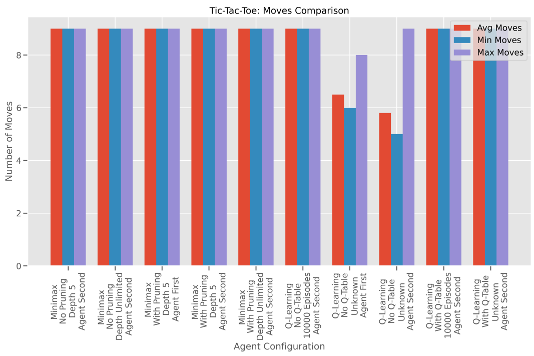

Figure 1: Moves comparison of Minimax and Q-Learning agents in Tic Tac Toe.

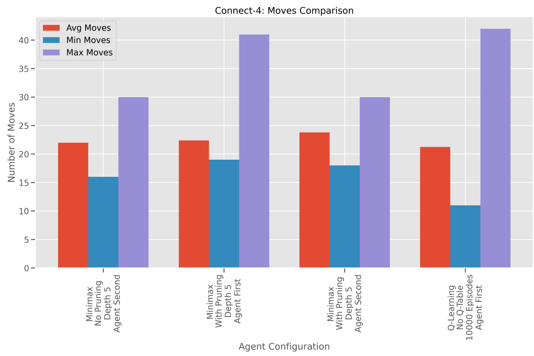

Figure 2: Moves comparison of Minimax and Q-Learning agents in Connect 4.

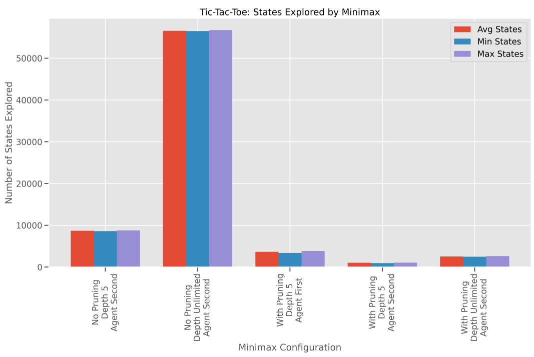

Figure 3: States explored comparison of Minimax and Q-Learning agents in Tic Tac Toe.

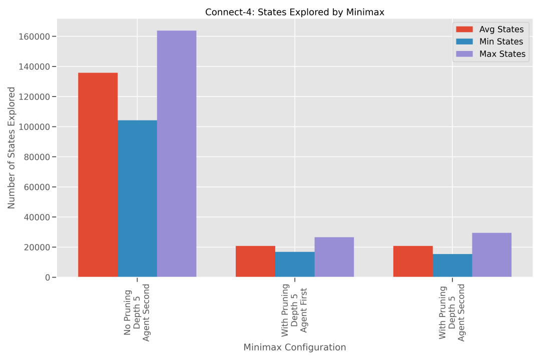

Figure 4: States explored comparison of Minimax and Q-Learning agents in Connect 4.

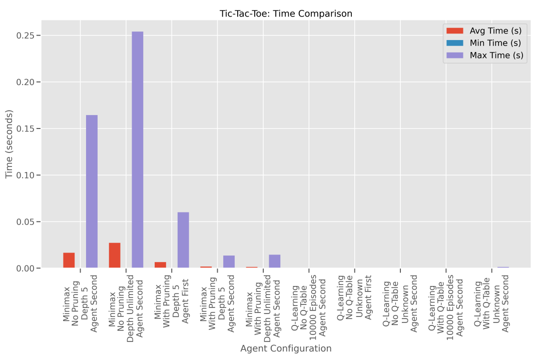

Figure 5: Time comparison of Minimax and Q-Learning agents in Tic Tac Toe.

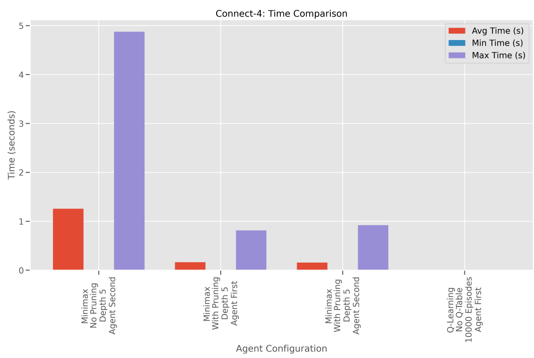

Figure 6: Time comparison of Minimax and Q-Learning agents in Connect 4.

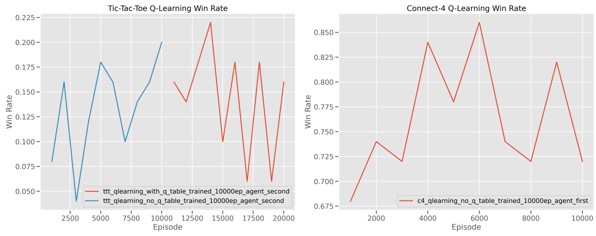

Figure 7: Q-Learning Win Rate During Training.

Tic Tac Toe

- **Moves & States:** The results indicate that in Tic Tac Toe both algorithms perform efficiently (that is both result in draw always), generally requiring 9 moves. When we do not train a Q-table, a random is employed, leading to a variable number of moves observed in that case (see 1). Minimax typically required fewer states to explore; however, in scenarios without pruning and depth limitations, the state exploration increased significantly. This suggests that for larger games, failing to implement these strategies can lead to highly inefficient agents (see 3).

- **Decision Time:** As reflected in 5, decision times are negligible for qlearning, which is expected given that after training, its just a lookup operation. However, for minimax, the decision time is higher when there is no pruning and depth limitations, which is expected given the high branching factor of Tic Tac Toe.

- **Overall Performance:** Minimax and Q-learning agents both result in draw always, which is expected given that the default semi-intelligent opponent is very optimal for tic tac toe.

Connect 4

- **Moves & States:** For Connect 4, the average moves required increase considerably, as shown in 2. The complexity is further highlighted by the vastly higher number of states explored (refer to 4), which aligns with the theoretical state space of approximately 1.4 x 10^{35} configurations. When there is no pruning, the state exploration is extremely high, which is expected given the high branching factor of Connect 4.

- **Decision Time:** The decision time for Connect 4 (presented in 6) is substantially longer, especially for Minimax. For qlearning, the decision time is negligible, which is expected given that after training, its just a lookup operation. This reinforces the need for heuristics such as depth-limiting or timeout parameters to keep the search tractable.

- **Learning and Adaptation:** In the case of Q-Learning, the training win/loss rate demonstrates a gradual convergence up until 13000 episodes, after which it fluctuates around 0.75 (see 7). It suggests that the can learn upto a certain point and then it plateaus and thus requires a better method than using a q-table.

- **Overall Performance:** Minimax always wins (due to its searching capabilities), which is expected given that the default opponent is a semi-intelligent and is optimal enough for connect 4. Whereas, Q-learning is generally able to win against the default opponent but losses quite a lot of games (about 22% of the games) (see table 1) due to its final state evaluation win rate being low (around 72%).

Algorithms vs Each Other

To assess the comparative strengths of the implemented algorithms, we conducted direct head-to-head matches between Minimax and Q-Learning agents in both Tic Tac Toe and Connect 4. Each experiment consisted of 100 games, alternating player roles to ensure fairness.

The Q-Learning agents used in these evaluations were pretrained (on semi-intelligent opponent from before) but not retrained or fine-tuned against the Minimax agent. This setup helps us assess how well Q-Learning generalizes to stronger, previously unseen opponents.

Game	1 Type	2 Type	1 Wins	2 Wins	Draws	Games Played	Avg. Moves (A1/A2)	Avg. Time/Move (A1/A2, s)	States Explored	Avg. Game Duration (s)
Connect 4	Q-Learning	Minimax	0	100	0	100	8.0 / 8.0	0.0024 / 0.5008	2,212,800	4.03
Connect 4	Minimax	Q-Learning	100	0	0	100	5.0 / 4.0	0.4702 / 0.0016	1,318,500	2.36
Tic Tac Toe	Q-Learning	Minimax	0	89	11	100	3.59 / 3.48	0.0000 / 0.0055	142,830	0.02
Tic Tac Toe	Minimax	Q-Learning	43	0	57	100	4.63 / 3.63	0.0122 / 0.0000	355,791	0.06

Table 2: Comparison of Performance in Connect 4 and Tic Tac Toe

- **Minimax Consistently Outperforms Q-Learning:** Across all configurations, Minimax decisively defeats the pretrained Q-Learning agent. This reflects the ability of search-based algorithms to consistently exploit suboptimal policies learned during training.
- **Generalization Gap in Q-Learning:** The Q-Learning agents, although pretrained, were not exposed to Minimax-like adversaries during training. As a result, they fail to generalize to these stronger opponents, especially visible in the 0% win rates.
- **Efficiency and Strategy:** Q-Learning agents respond quickly with near-zero time per move, but this comes at the cost of tactical depth. Minimax agents invest more time (e.g., 0.5s per move in Connect 4) and leverage evaluation functions to identify stronger positions.
- **Game-Specific Trends:** In Tic Tac Toe, the Minimax achieves a high number of draws when playing second, reflecting optimal counterplay. In Connect 4, its dominance is absolute due to the complexity of the board and greater decision space.

Overall Comparison of Algorithms

The evaluation of Minimax and Q-Learning agents across both Tic Tac Toe and Connect 4, using both default opponents and head-to-head comparisons, reveals distinct strengths, weaknesses, and trade-offs between search-based and learning-based approaches.

- **Performance Consistency:** Minimax consistently outperformed Q-Learning in all test scenarios, especially in head-to-head matches. It was able to exploit the weaknesses of the Q-Learning agent, even when the latter was pretrained. In both games, Minimax achieved either a perfect or near-perfect win rate when facing the default opponent or the Q-Learning agent.
- **Adaptability vs Optimality:** Q-Learning showed that it can learn effective policies against specific types of opponents, particularly in Connect 4, where its pretrained model achieved a win rate of 75% against the default agent. However, it failed to generalize to stronger adversaries like Minimax, which exploited its lack of foresight and positional reasoning.
- **Resource Efficiency:** Q-Learning is significantly faster at inference time due to its table-based policy lookup. This makes it more scalable for real-time or embedded use cases. Minimax, while slower, remains tractable in practice when enhanced with alpha-beta pruning and depth limitations, even in large state spaces like Connect 4. However, the computational cost becomes significant as the search depth increases or pruning is disabled.
- **Strategic Depth:** The results confirm that Minimax with pruning and evaluation heuristics provides robust performance even under constrained resources. In contrast, Q-Learning requires extensive training and cannot adapt on-the-fly to new strategies or agents without retraining.
- **Game-Specific Observations:** - In Tic Tac Toe, both algorithms tend to converge to draws against the optimal default opponent. This reinforces the idea that the game's small state space allows even basic agents to reach near-optimal play. - In Connect 4, Minimax consistently dominated, suggesting that its tactical foresight gives it a substantial edge in deeper games. Q-Learning's performance plateaus, indicating limited policy generalization despite training.
- **Engineering Considerations:** Implementing timeouts and depth constraints was essential for ensuring that Minimax remained usable in Connect 4. Without these, the would often run indefinitely. Conversely, the Q-Learning remained responsive throughout but incurred the upfront cost of long training times and storage for the Q-table.

While Minimax is computationally more intensive, it provides reliable, optimal play when appropriately constrained. Q-Learning, on the other hand, is more scalable and efficient at runtime, but is highly dependent on training quality and lacks adaptability to unfamiliar strategies. For environments with small to moderate state spaces or where deterministic performance is essential, Minimax is the preferred choice. For large-scale, stochastic, or dynamic environments where retraining is viable, Q-Learning may be more appropriate.

Conclusion

This report highlights the journey of developing, integrating, and evaluating AI agents for two popular board games, Tic Tac Toe and Connect 4, using Minimax and Q-Learning algorithms. The designs focused on modularity and extensibility, ensuring a clear separation of logic and interface, which facilitated smooth experimentation and integration of the agents.

The experimental results reveal that Minimax, especially when enhanced with alpha-beta pruning, delivered strong and consistent performance in both games, with particular success in Connect 4 where strategic depth plays a crucial role. On the other hand, Q-Learning proved effective in controlled scenarios and was computationally efficient during inference, but it faced challenges in generalizing against stronger opponents like Minimax, especially in unfamiliar situations.

The analysis brings to light significant trade-offs: Minimax shines in deterministic environments with manageable search spaces but can become resource-intensive in larger domains. In contrast, Q-Learning offers greater scalability during runtime and is better suited for dynamic environments, though it requires substantial training and lacks adaptability without retraining.

In summary, this project provided an engaging exploration of classical AI and reinforcement learning methods in adversarial game settings, showcasing their strengths, limitations, and practical considerations for implementation and deployment.

A Game Classes

A.1 Tic Tac Toe

```
import numpy as np
import pygame
import time
from enum import Enum

class TicTacToe:
    def __init__(self):
        self.board = np.zeros((3, 3), dtype=int)  # 0 for empty, 1 for X, 2 for O
        self.current_player = 1  # Player 1 (X) starts
        self.game_over = False
        self.winner = None

    def reset(self):
        """Reset the game to initial state."""
        self.board = np.zeros((3, 3), dtype=int)
        self.current_player = 1
        self.game_over = False
        self.winner = None
        return self.get_state()

    def get_state(self):
        """Return current state of the game."""
        return self.board.copy()

    def get_legal_moves(self):
        """Return list of legal moves as (row, col) tuples."""
        if self.game_over:
            return []
        return [(i, j) for i in range(3) for j in range(3) if self.board[i, j] == 0]

    def make_move(self, move):
        """Make a move on the board.

        Args:
            move: tuple (row, col)

        Returns:
            bool: True if the move was valid, False otherwise
        """
        row, col = move
        if self.game_over or row < 0 or row > 2 or col < 0 or col > 2 or self.board[row, col] !=
             0:
            return False

        self.board[row, col] = self.current_player
        self._check_game_over()
        self.current_player = 3 - self.current_player  # Switch players (1 -> 2, 2 -> 1)
        return True

    def _check_game_over(self):
        """Check if the game is over (win or draw)."""
        # Check rows
        for row in range(3):
            if self.board[row, 0] != 0 and self.board[row, 0] == self.board[row, 1] ==
                 self.board[row, 2]:
                self.game_over = True
                self.winner = self.board[row, 0]
                return

        # Check columns
        for col in range(3):
```

```
            if self.board[0, col] != 0 and self.board[0, col] == self.board[1, col] ==
                 self.board[2, col]:
                self.game_over = True
                self.winner = self.board[0, col]
                return

        # Check diagonals
        if self.board[0, 0] != 0 and self.board[0, 0] == self.board[1, 1] == self.board[2, 2]:
            self.game_over = True
            self.winner = self.board[0, 0]
            return

        if self.board[0, 2] != 0 and self.board[0, 2] == self.board[1, 1] == self.board[2, 0]:
            self.game_over = True
            self.winner = self.board[0, 2]
            return

        # Check for draw
        if np.all(self.board != 0):
            self.game_over = True
            self.winner = 0  # Draw
            return

    def is_game_over(self):
        """Return whether the game is over."""
        return self.game_over

    def get_winner(self):
        """Return the winner (1 for X, 2 for O, 0 for draw, None if game not over)."""
        return self.winner

class PlayerType(Enum):
    HUMAN = 0
    AI = 1
    SEMI_INTELLIGENT = 2

class GameMode(Enum):
    HUMAN_VS_HUMAN = 0
    HUMAN_VS_AI = 1
    HUMAN_VS_SEMI = 2
    AI_VS_SEMI = 3
    AI_VS_AI = 4

class TicTacToeUI:
    def __init__(self):
        pygame.init()
        self.game = TicTacToe()
        self.width, self.height = 950, 950
        self.screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption("Tic Tac Toe")

        # Colors
        self.bg_color = (240, 240, 240)
        self.line_color = (80, 80, 80)
        self.x_color = (66, 134, 244)
        self.o_color = (255, 87, 87)
        self.text_color = (50, 50, 50)
        self.highlight_color = (180, 180, 180)

        # Size and positions
        self.board_size = 450
        self.cell_size = self.board_size // 3
        self.board_margin = (self.width - self.board_size) // 2
        self.line_width = 4

        # Fonts
        self.font = pygame.font.SysFont('Arial', 30)
        self.big_font = pygame.font.SysFont('Arial', 50)

        # Game mode and players
        self.game_mode = GameMode.HUMAN_VS_HUMAN
        self.player1_type = PlayerType.HUMAN
        self.player2_type = PlayerType.HUMAN
        self.player1_agent = None  # AI agent for player 1
        self.player2_agent = None  # AI agent for player 2
        self.player_move = True  # True if current turn is for human input
        self.ai_move_delay = 0.5  # Delay between AI moves in seconds
        self.last_ai_move_time = 0  # Last time AI made a move

    def set_game_mode(self, mode):
        """Set the game mode and initialize appropriate players."""
        self.game_mode = mode

        if mode == GameMode.HUMAN_VS_HUMAN:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.HUMAN
            self.player_move = True

        elif mode == GameMode.HUMAN_VS_AI:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.AI
```

```
            self.player_move = True

        elif mode == GameMode.HUMAN_VS_SEMI:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.SEMI_INTELLIGENT
            self.player_move = True

        elif mode == GameMode.AI_VS_SEMI:
            self.player1_type = PlayerType.AI
            self.player2_type = PlayerType.SEMI_INTELLIGENT
            self.player_move = False

        elif mode == GameMode.AI_VS_AI:
            # Both players are AI agents
            self.player1_type = PlayerType.AI
            self.player2_type = PlayerType.AI
            self.player_move = False

        # Reset the game
        self.game.reset()

    def set_player1_agent(self, agent):
        """Set AI agent for player 1."""
        self.player1_agent = agent

    def set_player2_agent(self, agent):
        """Set AI agent for player 2."""
        self.player2_agent = agent

    def run(self):
        """Main game loop."""
        running = True

        while running:
            current_time = time.time()

            # Handle events
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False

                if event.type == pygame.MOUSEBUTTONDOWN:
                    # Human move handling
                    if self.player_move and not self.game.is_game_over():
                        self._handle_click(event.pos)

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_r:  # Reset game
                        self.game.reset()
                        self.player_move = (self.player1_type == PlayerType.HUMAN)

            # Handle AI agent moves
            if not self.game.is_game_over():
                current_player_num = self.game.current_player

                # Player 1's turn (X)
                if current_player_num == 1:
                    if self.player1_type in [PlayerType.AI, PlayerType.SEMI_INTELLIGENT] and
                        (current_time - self.last_ai_move_time >= self.ai_move_delay):
                        agent = self.player1_agent
                        if agent and hasattr(agent, 'get_move'):
                            move = agent.get_move(self.game.get_state())
                            if move:
                                self.game.make_move(move)
                                self.last_ai_move_time = current_time
                                # After AI move, it could be human's turn
                                self.player_move = (self.player2_type == PlayerType.HUMAN)

                # Player 2's turn (O)
                else:
                    if self.player2_type in [PlayerType.AI, PlayerType.SEMI_INTELLIGENT] and
                        (current_time - self.last_ai_move_time >= self.ai_move_delay):
                        agent = self.player2_agent
                        if agent and hasattr(agent, 'get_move'):
                            move = agent.get_move(self.game.get_state())
                            if move:
                                self.game.make_move(move)
                                self.last_ai_move_time = current_time
                                # After AI move, it could be human's turn
                                self.player_move = (self.player1_type == PlayerType.HUMAN)

            # Draw everything
            self._draw()
            pygame.display.flip()

            # Small delay to avoid high CPU usage
            time.sleep(0.01)

        pygame.quit()

    def _handle_click(self, pos):
```

```
        """Handle mouse click to make a move."""
        x, y = pos

        # Check if click is within the board
        if (self.board_margin <= x <= self.board_margin + self.board_size and
            self.board_margin <= y <= self.board_margin + self.board_size):

            # Convert click position to grid indices
            row = (y - self.board_margin) // self.cell_size
            col = (x - self.board_margin) // self.cell_size

            # Make move if valid
            if self.game.make_move((row, col)):
                # After human move, determine who plays next
                current_player_num = self.game.current_player
                if current_player_num == 1:
                    self.player_move = (self.player1_type == PlayerType.HUMAN)
                else:
                    self.player_move = (self.player2_type == PlayerType.HUMAN)

    def _draw(self):
        """Draw the game board and UI."""
        # Fill background
        self.screen.fill(self.bg_color)

        # Draw title and status
        title = self.big_font.render("Tic Tac Toe", True, self.text_color)
        self.screen.blit(title, (self.width // 2 - title.get_width() // 2, 20))

        # Display current game mode
        mode_names = {
            GameMode.HUMAN_VS_HUMAN: "Human vs Human",
            GameMode.HUMAN_VS_AI: "Human vs AI",
            GameMode.HUMAN_VS_SEMI: "Human vs Semi-Intelligent",
            GameMode.AI_VS_SEMI: "AI vs Semi-Intelligent",
            GameMode.AI_VS_AI: "AI vs AI"
        }
        mode_text = self.font.render(f"Mode: {mode_names[self.game_mode]}", True, self.text_color)
        self.screen.blit(mode_text, (self.width // 2 - mode_text.get_width() // 2, 80))

        # Game status
        if self.game.is_game_over():
            if self.game.get_winner() == 1:
                status = self.font.render("X wins!", True, self.x_color)
            elif self.game.get_winner() == 2:
                status = self.font.render("O wins!", True, self.o_color)
            else:
                status = self.font.render("Draw!", True, self.text_color)

            restart = self.font.render("Press R to restart", True, self.text_color)
            self.screen.blit(restart, (self.width // 2 - restart.get_width() // 2, self.height -
                 50))
        else:
            current_player_text = "X's turn" if self.game.current_player == 1 else "O's turn"
            status = self.font.render(current_player_text, True, self.x_color if
                 self.game.current_player == 1 else self.o_color)

        self.screen.blit(status, (self.width // 2 - status.get_width() // 2, self.height - 100))

        # Draw board
        for i in range(4):
            # Horizontal lines
            pygame.draw.line(
                self.screen,
                self.line_color,
                (self.board_margin, self.board_margin + i * self.cell_size),
                (self.board_margin + self.board_size, self.board_margin + i * self.cell_size),
                self.line_width
            )

            # Vertical lines
            pygame.draw.line(
                self.screen,
                self.line_color,
                (self.board_margin + i * self.cell_size, self.board_margin),
                (self.board_margin + i * self.cell_size, self.board_margin + self.board_size),
                self.line_width
            )

        # Draw X's and O's
        for row in range(3):
            for col in range(3):
                center_x = self.board_margin + col * self.cell_size + self.cell_size // 2
                center_y = self.board_margin + row * self.cell_size + self.cell_size // 2

                if self.game.board[row, col] == 1:  # X
                    size = self.cell_size // 2 - 20
                    pygame.draw.line(
                        self.screen,
                        self.x_color,
                        (center_x - size, center_y - size),
```

```
                        (center_x + size, center_y + size),
                        self.line_width + 3
                    )
                    pygame.draw.line(
                        self.screen,
                        self.x_color,
                        (center_x - size, center_y + size),
                        (center_x + size, center_y - size),
                        self.line_width + 3
                    )

                elif self.game.board[row, col] == 2:  # O
                    size = self.cell_size // 2 - 15
                    pygame.draw.circle(
                        self.screen,
                        self.o_color,
                        (center_x, center_y),
                        size,
                        self.line_width + 3
                    )
```

A.2 Connect 4

```
import numpy as np
import pygame
import time
from enum import Enum
from games.tic_tac_toe import PlayerType, GameMode

class Connect4:
    def __init__(self):
        self.rows = 6
        self.cols = 7
        self.board = np.zeros((self.rows, self.cols), dtype=int)  # 0 for empty, 1 and 2 for
            players
        self.current_player = 1  # Player 1 starts
        self.game_over = False
        self.winner = None
        self.last_move = None

    def reset(self):
        """Reset the game to initial state."""
        self.board = np.zeros((self.rows, self.cols), dtype=int)
        self.current_player = 1
        self.game_over = False
        self.winner = None
        self.last_move = None
        return self.get_state()

    def get_state(self):
        """Return current state of the game."""
        return self.board.copy()

    def get_legal_moves(self):
        """Return list of legal moves (columns where a piece can be dropped)."""
        if self.game_over:
            return []
        return [col for col in range(self.cols) if self.board[0, col] == 0]

    def make_move(self, col):
        """Make a move by dropping a piece in the specified column.

        Args:
            col: Column to drop the piece

        Returns:
            bool: True if the move was valid, False otherwise
        """
        if self.game_over or col < 0 or col >= self.cols or self.board[0, col] != 0:
            return False

        # Find the lowest empty row in the selected column
        for row in range(self.rows - 1, -1, -1):
            if self.board[row, col] == 0:
                self.board[row, col] = self.current_player
                self.last_move = (row, col)
                break

        self._check_game_over()
        self.current_player = 3 - self.current_player  # Switch players (1 -> 2, 2 -> 1)
        return True

    def _check_game_over(self):
        """Check if the game is over (win or draw)."""
        if self.last_move is None:
            return

        row, col = self.last_move
```

```
        player = self.board[row, col]

        # Check horizontal
        for c in range(max(0, col - 3), min(col + 1, self.cols - 3)):
            if self.board[row, c] == player and self.board[row, c+1] == player \
               and self.board[row, c+2] == player and self.board[row, c+3] == player:
                self.game_over = True
                self.winner = player
                return

        # Check vertical
        for r in range(max(0, row - 3), min(row + 1, self.rows - 3)):
            if self.board[r, col] == player and self.board[r+1, col] == player \
               and self.board[r+2, col] == player and self.board[r+3, col] == player:
                self.game_over = True
                self.winner = player
                return

        # Check diagonal (positive slope)
        for r, c in zip(range(row, max(row-4, -1), -1), range(col, max(col-4, -1), -1)):
            if r+3 < self.rows and c+3 < self.cols:
                if self.board[r, c] == player and self.board[r+1, c+1] == player \
                   and self.board[r+2, c+2] == player and self.board[r+3, c+3] == player:
                    self.game_over = True
                    self.winner = player
                    return

        # Check diagonal (negative slope)
        for r, c in zip(range(row, max(row-4, -1), -1), range(col, min(col+4, self.cols))):
            if r+3 < self.rows and c-3 >= 0:
                if self.board[r, c] == player and self.board[r+1, c-1] == player \
                   and self.board[r+2, c-2] == player and self.board[r+3, c-3] == player:
                    self.game_over = True
                    self.winner = player
                    return

        # Check for draw
        if np.all(self.board[0, :] != 0):
            self.game_over = True
            self.winner = 0  # Draw
            return

    def is_game_over(self):
        """Return whether the game is over."""
        return self.game_over

    def get_winner(self):
        """Return the winner (1 or 2 for players, 0 for draw, None if game not over)."""
        return self.winner

class Connect4UI:
    def __init__(self):
        pygame.init()
        self.game = Connect4()
        self.cell_size = 80
        self.width = self.game.cols * self.cell_size
        self.height = (self.game.rows + 1) * self.cell_size + 200  # Extra space for UI
        self.screen = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption("Connect 4")

        # Colors
        self.bg_color = (0, 105, 148)  # Blue background
        self.board_color = (0, 65, 118)
        self.player1_color = (255, 51, 51)  # Red
        self.player2_color = (255, 236, 51)  # Yellow
        self.text_color = (0, 0, 0)
        self.highlight_color = (0, 180, 215)

        # Fonts
        self.font = pygame.font.SysFont('Arial', 30)
        self.big_font = pygame.font.SysFont('Arial', 40)

        # Game mode and players
        self.game_mode = GameMode.HUMAN_VS_HUMAN
        self.player1_type = PlayerType.HUMAN
        self.player2_type = PlayerType.HUMAN
        self.player1_agent = None  # AI agent for player 1
        self.player2_agent = None  # AI agent for player 2
        self.player_move = True  # True if current turn is for human input
        self.ai_move_delay = 0.5  # Delay between AI moves in seconds
        self.last_ai_move_time = 0  # Last time AI made a move

        # Mode names dictionary for UI
        self.mode_names = {
            GameMode.HUMAN_VS_HUMAN: "Human vs Human",
            GameMode.HUMAN_VS_AI: "Human vs AI",
            GameMode.HUMAN_VS_SEMI: "Human vs Semi-Intelligent",
            GameMode.AI_VS_SEMI: "AI vs Semi-Intelligent",
            GameMode.AI_VS_AI: "AI vs AI"
        }
```

```
        # Animation
        self.anim_active = False
        self.anim_col = 0
        self.anim_row = 0
        self.anim_y = 0
        self.anim_speed = 15
        self.anim_player = 0

    def set_game_mode(self, mode):
        """Set the game mode and initialize appropriate players."""
        if isinstance(mode, int):
            mode = GameMode(mode)  # Convert int to GameMode Enum
        self.game_mode = mode

        if mode == GameMode.HUMAN_VS_HUMAN:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.HUMAN
            self.player_move = True

        elif mode == GameMode.HUMAN_VS_AI:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.AI
            self.player_move = True

        elif mode == GameMode.HUMAN_VS_SEMI:
            self.player1_type = PlayerType.HUMAN
            self.player2_type = PlayerType.SEMI_INTELLIGENT
            self.player_move = True

        elif mode == GameMode.AI_VS_SEMI:
            self.player1_type = PlayerType.AI
            self.player2_type = PlayerType.SEMI_INTELLIGENT
            self.player_move = False

        elif mode == GameMode.AI_VS_AI:
            # Both players are AI agents
            self.player1_type = PlayerType.AI
            self.player2_type = PlayerType.AI
            self.player_move = False

        # Reset the game
        self.game.reset()
        self.anim_active = False

    def set_player1_agent(self, agent):
        """Set AI agent for player 1."""
        self.player1_agent = agent

    def set_player2_agent(self, agent):
        """Set AI agent for player 2."""
        self.player2_agent = agent

    def run(self):
        """Main game loop."""
        running = True

        while running:
            current_time = time.time()

            # Handle events
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    running = False

                if event.type == pygame.MOUSEBUTTONDOWN:
                    # Human move handling
                    if self.player_move and not self.game.is_game_over() and not self.anim_active:
                        self._handle_click(event.pos)

                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_r:  # Reset game
                        self.game.reset()
                        self.player_move = (self.player1_type == PlayerType.HUMAN)
                        self.anim_active = False

            # Handle AI and semi-intelligent agent moves
            if not self.game.is_game_over() and not self.anim_active:
                current_player_num = self.game.current_player

                # Player 1's turn (Red)
                if current_player_num == 1:
                    if self.player1_type in [PlayerType.AI, PlayerType.SEMI_INTELLIGENT] and
                        (current_time - self.last_ai_move_time >= self.ai_move_delay):
                        agent = self.player1_agent
                        if agent and hasattr(agent, 'get_move'):
                            move = agent.get_move(self.game.get_state())
                            if move is not None:
                                self._start_animation(move)
                                self.last_ai_move_time = current_time

                # Player 2's turn (Yellow)
```

```
            else:
                if self.player2_type in [PlayerType.AI, PlayerType.SEMI_INTELLIGENT] and
                    (current_time - self.last_ai_move_time >= self.ai_move_delay):
                    agent = self.player2_agent
                    if agent and hasattr(agent, 'get_move'):
                        move = agent.get_move(self.game.get_state())
                        if move is not None:
                            self._start_animation(move)
                            self.last_ai_move_time = current_time

        # Update animation
        if self.anim_active:
            self._update_animation()

        # Draw everything
        self._draw()
        pygame.display.flip()

        # Small delay to avoid high CPU usage
        time.sleep(0.01)

    pygame.quit()

def _start_animation(self, col):
    """Start animation for dropping a piece."""
    if col in self.game.get_legal_moves():
        self.anim_active = True
        self.anim_col = col
        self.anim_row = 0
        for row in range(self.game.rows - 1, -1, -1):
            if self.game.board[row, col] == 0:
                self.anim_row = row
                break
        self.anim_y = self.cell_size
        self.anim_player = self.game.current_player

def _update_animation(self):
    """Update dropping animation."""
    target_y = (self.anim_row + 1) * self.cell_size

    if self.anim_y < target_y:
        self.anim_y += self.anim_speed
    else:
        self.anim_active = False
        self.game.make_move(self.anim_col)

        # After a piece is dropped, determine who plays next
        current_player = self.game.current_player
        if current_player == 1:
            self.player_move = (self.player1_type == PlayerType.HUMAN)
        else:
            self.player_move = (self.player2_type == PlayerType.HUMAN)

def _handle_click(self, pos):
    """Handle mouse click to make a move."""
    x, y = pos

    # Check if click is within the valid area (above the board)
    if y < self.cell_size and 0 <= x < self.width:
        col = x // self.cell_size
        if col in self.game.get_legal_moves():
            self._start_animation(col)

def _draw(self):
    """Draw the game board and UI."""
    # Fill background
    self.screen.fill(self.bg_color)

    # Draw title and game mode info
    if self.game.is_game_over():
        if self.game.get_winner() == 1:
            status = self.big_font.render("Red wins!", True, self.player1_color)
        elif self.game.get_winner() == 2:
            status = self.big_font.render("Yellow wins!", True, self.player2_color)
        else:
            status = self.big_font.render("Draw!", True, self.text_color)

        restart = self.font.render("Press R to restart", True, self.text_color)
        self.screen.blit(restart, (self.width // 2 - restart.get_width() // 2, 20))
    else:
        if self.game.current_player == 1:
            status = self.big_font.render("Red's turn", True, self.player1_color)
        else:
            status = self.big_font.render("Yellow's turn", True, self.player2_color)

    self.screen.blit(status, (self.width // 2 - status.get_width() // 2, self.height - 160))

    # Display current game mode
    mode_text = self.font.render(f"Mode: {self.mode_names[self.game_mode]}", True,
        self.text_color)
    self.screen.blit(mode_text, (30, self.height - 50))
```

```python
        # Draw board background
        pygame.draw.rect(
            self.screen,
            self.board_color,
            (0, self.cell_size, self.width, self.game.rows * self.cell_size)
        )

        # Draw empty slots and pieces
        for row in range(self.game.rows):
            for col in range(self.game.cols):
                center_x = col * self.cell_size + self.cell_size // 2
                center_y = (row + 1) * self.cell_size + self.cell_size // 2

                # Draw empty slot
                pygame.draw.circle(
                    self.screen,
                    self.bg_color,
                    (center_x, center_y),
                    self.cell_size // 2 - 5
                )

                # Draw piece
                if self.game.board[row, col] == 1:
                    pygame.draw.circle(
                        self.screen,
                        self.player1_color,
                        (center_x, center_y),
                        self.cell_size // 2 - 5
                    )
                elif self.game.board[row, col] == 2:
                    pygame.draw.circle(
                        self.screen,
                        self.player2_color,
                        (center_x, center_y),
                        self.cell_size // 2 - 5
                    )

        # Draw animation
        if self.anim_active:
            center_x = self.anim_col * self.cell_size + self.cell_size // 2
            center_y = self.anim_y

            color = self.player1_color if self.anim_player == 1 else self.player2_color
            pygame.draw.circle(
                self.screen,
                color,
                (center_x, center_y),
                self.cell_size // 2 - 5
            )

        # Draw column highlight on hover (only when it's human's turn)
        if not self.game.is_game_over() and not self.anim_active and self.player_move:
            mouse_x, mouse_y = pygame.mouse.get_pos()
            if mouse_y < self.cell_size:
                col = mouse_x // self.cell_size
                if 0 <= col < self.game.cols and col in self.game.get_legal_moves():
                    pygame.draw.rect(
                        self.screen,
                        self.highlight_color,
                        (col * self.cell_size, 0, self.cell_size, self.cell_size),
                        3
                    )

                    # Draw preview piece
                    color = self.player1_color if self.game.current_player == 1 else
                        self.player2_color
                    pygame.draw.circle(
                        self.screen,
                        color,
                        (col * self.cell_size + self.cell_size // 2, self.cell_size // 2),
                        self.cell_size // 2 - 5
                    )
```

B Minimax Implementation

B.1 Minimax Base

```python
import time
import numpy as np
from abc import ABC, abstractmethod

class MinimaxBase(ABC):
    """
    Abstract base class for minimax algorithm implementations.
    Game-specific implementations should inherit from this class and implement
```

```
    the abstract methods.
    """

    def __init__(self, max_depth=float('inf'), metrics_manager=None, use_pruning=True):
        """
        Initialize the MinimaxBase with configurable parameters.

        Args:
            max_depth (int): Maximum depth for the minimax algorithm (default: infinity)
            metrics_manager: Optional metrics manager for tracking performance
            use_pruning (bool): Whether to use alpha-beta pruning (default: True)
        """
        self.max_depth = max_depth
        self.metrics_manager = metrics_manager
        self.player = None  # Will be set when get_move is called
        self.use_pruning = use_pruning
        self.states_explored = 0  # Counter for states explored

    def get_move(self, state):
        """
        Get the best move for the current state using minimax algorithm.

        Args:
            state: Current state of the game

        Returns:
            The best move according to minimax algorithm
        """
        self.player = self._get_current_player(state)
        self.states_explored = 0  # Reset counter

        if self.use_pruning:
            best_move = self._find_best_move_with_pruning(state)
        else:
            best_move = self._find_best_move_without_pruning(state)

        # Record states explored if metrics manager is available
        if self.metrics_manager:
            self.metrics_manager.record_states_explored(self.states_explored)

        return best_move

    def _find_best_move_with_pruning(self, state):
        """
        Find the best move using minimax algorithm with alpha-beta pruning.

        Args:
            state: Current state of the game

        Returns:
            The best move according to minimax algorithm
        """
        best_val = float('-inf')
        best_move = None
        alpha = float('-inf')
        beta = float('inf')

        for move in self._get_legal_moves(state):
            # Make the move
            new_state = self._make_move(state.copy(), move, self.player)

            # Calculate value for this move
            move_val = self._minimax_with_pruning(new_state, self.max_depth - 1, False, alpha,
                 beta)

            # Update best move if this is better
            if move_val > best_val:
                best_val = move_val
                best_move = move

            # Update alpha
            alpha = max(alpha, best_val)

        return best_move

    def _find_best_move_without_pruning(self, state):
        """
        Find the best move using minimax algorithm without alpha-beta pruning.

        Args:
            state: Current state of the game

        Returns:
            The best move according to minimax algorithm
        """
        best_val = float('-inf')
        best_move = None

        for move in self._get_legal_moves(state):
            # Make the move
            new_state = self._make_move(state.copy(), move, self.player)
```

```python
            # Calculate value for this move
            move_val = self._minimax_without_pruning(new_state, self.max_depth - 1, False)

            # Update best move if this is better
            if move_val > best_val:
                best_val = move_val
                best_move = move

        return best_move

    def _minimax_with_pruning(self, state, depth, is_maximizing, alpha, beta):
        """
        Minimax algorithm with alpha-beta pruning.

        Args:
            state: Current state of the game
            depth (int): Current depth in the search tree
            is_maximizing (bool): True if current player is maximizing, False otherwise
            alpha: Alpha value for pruning
            beta: Beta value for pruning

        Returns:
            The best score for the current state
        """
        # Increment state exploration counter
        self.states_explored += 1

        # Check if we've reached a terminal state
        winner = self._check_winner(state)
        if winner is not None:
            return self._evaluate_terminal(winner)

        # Check if we've reached maximum depth
        if depth == 0:
            return self._evaluate_board(state)

        # Get current player
        current_player = self._get_player(is_maximizing)

        # Maximizing player
        if is_maximizing:
            max_eval = float('-inf')
            for move in self._get_legal_moves(state):
                new_state = self._make_move(state.copy(), move, current_player)
                eval = self._minimax_with_pruning(new_state, depth - 1, False, alpha, beta)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break  # Beta cutoff
            return max_eval

        # Minimizing player
        else:
            min_eval = float('inf')
            for move in self._get_legal_moves(state):
                new_state = self._make_move(state.copy(), move, current_player)
                eval = self._minimax_with_pruning(new_state, depth - 1, True, alpha, beta)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break  # Alpha cutoff
            return min_eval

    def _minimax_without_pruning(self, state, depth, is_maximizing):
        """
        Minimax algorithm without alpha-beta pruning.

        Args:
            state: Current state of the game
            depth (int): Current depth in the search tree
            is_maximizing (bool): True if current player is maximizing, False otherwise

        Returns:
            The best score for the current state
        """
        # Increment state exploration counter
        self.states_explored += 1

        # Check if we've reached a terminal state
        winner = self._check_winner(state)
        if winner is not None:
            return self._evaluate_terminal(winner)

        # Check if we've reached maximum depth
        if depth == 0:
            return self._evaluate_board(state)

        # Get current player
        current_player = self._get_player(is_maximizing)
```

```python
        # Get legal moves
        legal_moves = self._get_legal_moves(state)

        # If no legal moves, it's a draw
        if not legal_moves:
            return 0

        # Maximizing player
        if is_maximizing:
            max_eval = float('-inf')
            for move in legal_moves:
                new_state = self._make_move(state.copy(), move, current_player)
                eval = self._minimax_without_pruning(new_state, depth - 1, False)
                max_eval = max(max_eval, eval)
            return max_eval

        # Minimizing player
        else:
            min_eval = float('inf')
            for move in legal_moves:
                new_state = self._make_move(state.copy(), move, current_player)
                eval = self._minimax_without_pruning(new_state, depth - 1, True)
                min_eval = min(min_eval, eval)
            return min_eval

    def _get_player(self, is_maximizing):
        """
        Get the player ID based on whether it's a maximizing or minimizing move.

        Args:
            is_maximizing (bool): True if it's a maximizing move

        Returns:
            The player ID
        """
        if is_maximizing:
            return self.player
        else:
            return 3 - self.player  # Switch between 1 and 2

    @abstractmethod
    def _get_current_player(self, state):
        """
        Get the current player from the state.

        Args:
            state: Current state of the game

        Returns:
            The current player ID
        """
        pass

    @abstractmethod
    def _get_legal_moves(self, state):
        """
        Get legal moves for the current state.

        Args:
            state: Current state of the game

        Returns:
            List of legal moves
        """
        pass

    @abstractmethod
    def _make_move(self, state, move, player):
        """
        Make a move on the board.

        Args:
            state: Current state of the game
            move: Move to make
            player: Player making the move

        Returns:
            New state after making the move
        """
        pass

    @abstractmethod
    def _check_winner(self, state):
        """
        Check if there's a winner in the current state.

        Args:
            state: Current state of the game

        Returns:
            The winner (1 or 2), 0 for draw, None if game is not over
```

```
        """
        pass

    @abstractmethod
    def _evaluate_board(self, state):
        """
        Evaluate the current board state.

        Args:
            state: Current state of the game

        Returns:
            Numerical score for the board state
        """
        pass

    @abstractmethod
    def _evaluate_terminal(self, winner):
        """
        Evaluate a terminal state.

        Args:
            winner: The winner (1 or 2), 0 for draw

        Returns:
            Numerical score for the terminal state
        """
        pass
```

B.2 Minimax Tic Tac Toe

```
import numpy as np
from .minimax_base import MinimaxBase

class MinimaxTicTacToe(MinimaxBase):
    """
    Minimax implementation for Tic-Tac-Toe.
    """

    def __init__(self, max_depth=9, metrics_manager=None, use_pruning=True):
        """
        Initialize the MinimaxTicTacToe with configurable parameters.
        Default max_depth is 9 since Tic-Tac-Toe has at most 9 moves.

        Args:
            max_depth (int): Maximum depth for the minimax algorithm
            metrics_manager: Optional metrics manager for tracking performance
            use_pruning (bool): Whether to use alpha-beta pruning
        """
        super().__init__(max_depth, metrics_manager, use_pruning)

    def _get_current_player(self, state):
        """
        Get the current player from the state.
        In Tic-Tac-Toe, we determine current player by counting pieces.

        Args:
            state: Current state of the game (numpy array)

        Returns:
            The current player ID (1 or 2)
        """
        # Count number of each player's pieces
        p1_count = np.count_nonzero(state == 1)
        p2_count = np.count_nonzero(state == 2)

        # Player 1 goes first, so if counts are equal, it's player 1's turn
        return 1 if p1_count <= p2_count else 2

    def _get_legal_moves(self, state):
        """
        Get legal moves for the current state.
        In Tic-Tac-Toe, legal moves are empty cells.

        Args:
            state: Current state of the game

        Returns:
            List of legal moves as (row, col) tuples
        """
        return [(i, j) for i in range(3) for j in range(3) if state[i, j] == 0]

    def _make_move(self, state, move, player):
        """
        Make a move on the board.

        Args:
            state: Current state of the game
```

```
            move: Move to make as (row, col)
            player: Player making the move

        Returns:
            New state after making the move
        """
        row, col = move
        state[row, col] = player
        return state

    def _check_winner(self, state):
        """
        Check if there's a winner in the current state.

        Args:
            state: Current state of the game

        Returns:
            The winner (1 or 2), 0 for draw, None if game is not over
        """
        # Check rows
        for row in range(3):
            if state[row, 0] != 0 and state[row, 0] == state[row, 1] == state[row, 2]:
                return state[row, 0]

        # Check columns
        for col in range(3):
            if state[0, col] != 0 and state[0, col] == state[1, col] == state[2, col]:
                return state[0, col]

        # Check diagonals
        if state[0, 0] != 0 and state[0, 0] == state[1, 1] == state[2, 2]:
            return state[0, 0]

        if state[0, 2] != 0 and state[0, 2] == state[1, 1] == state[2, 0]:
            return state[0, 2]

        # Check for draw (all cells filled)
        if np.all(state != 0):
            return 0  # Draw

        # Game not over yet
        return None

    def _evaluate_board(self, state):
        """
        Evaluate the current board state using a heuristic.
        For Tic-Tac-Toe, we'll use a simple scoring system based on potential wins.

        Args:
            state: Current state of the game

        Returns:
            Numerical score for the board state
        """
        score = 0

        # Check rows, columns, diagonals for potential wins
        # Add points for our potential wins, subtract for opponent's

        # Check rows
        for row in range(3):
            score += self._evaluate_line(state[row, :])

        # Check columns
        for col in range(3):
            score += self._evaluate_line(state[:, col])

        # Check diagonals
        score += self._evaluate_line(np.array([state[0, 0], state[1, 1], state[2, 2]]))
        score += self._evaluate_line(np.array([state[0, 2], state[1, 1], state[2, 0]]))

        return score

    def _evaluate_line(self, line):
        """
        Evaluate a line (row, column, or diagonal) for potential wins.

        Args:
            line: Array representing a line on the board

        Returns:
            Score for this line
        """
        our_pieces = np.count_nonzero(line == self.player)
        opponent_pieces = np.count_nonzero(line == (3 - self.player))
        empty_cells = np.count_nonzero(line == 0)

        # If we have a potential win (all our pieces or empty)
        if opponent_pieces == 0:
            if our_pieces == 2 and empty_cells == 1:  # Almost win
```

```
                return 10
            elif our_pieces == 1 and empty_cells == 2:  # Potential future win
                return 1

        # If opponent has a potential win (all their pieces or empty)
        if our_pieces == 0:
            if opponent_pieces == 2 and empty_cells == 1:  # Opponent almost win
                return -10
            elif opponent_pieces == 1 and empty_cells == 2:  # Opponent potential future win
                return -1

        return 0

    def _evaluate_terminal(self, winner):
        """
        Evaluate a terminal state.

        Args:
            winner: The winner (1 or 2), 0 for draw

        Returns:
            Numerical score for the terminal state
        """
        if winner == 0:  # Draw
            return 0
        elif winner == self.player:  # We win
            return 100
        else:  # Opponent wins
            return -100
```

B.3 Minimax Connect 4

```
    import numpy as np
from .minimax_base import MinimaxBase

class MinimaxConnect4(MinimaxBase):
    """
    Minimax implementation for Connect-4.
    """

    def __init__(self, max_depth=4, metrics_manager=None, use_pruning=True):
        """
        Initialize the MinimaxConnect4 with configurable parameters.
        Default max_depth is 4 for Connect-4 due to its branching factor.

        Args:
            max_depth (int): Maximum depth for the minimax algorithm
            metrics_manager: Optional metrics manager for tracking performance
            use_pruning (bool): Whether to use alpha-beta pruning
        """
        super().__init__(max_depth, metrics_manager, use_pruning)
        self.rows = 6
        self.cols = 7

    def _get_current_player(self, state):
        """
        Get the current player from the state.
        In Connect-4, we determine current player by counting pieces.

        Args:
            state: Current state of the game (numpy array)

        Returns:
            The current player ID (1 or 2)
        """
        # Count number of each player's pieces
        p1_count = np.count_nonzero(state == 1)
        p2_count = np.count_nonzero(state == 2)

        # Player 1 goes first, so if counts are equal, it's player 1's turn
        return 1 if p1_count <= p2_count else 2

    def _get_legal_moves(self, state):
        """
        Get legal moves for the current state.
        In Connect-4, legal moves are columns that aren't filled.

        Args:
            state: Current state of the game

        Returns:
            List of legal moves (column indices)
        """
        return [col for col in range(self.cols) if state[0, col] == 0]

    def _make_move(self, state, move, player):
        """
        Make a move on the board.
```

```
        Args:
            state: Current state of the game
            move: Move to make (column index)
            player: Player making the move

        Returns:
            New state after making the move
        """
        col = move

        # Find the lowest empty row in the selected column
        for row in range(self.rows - 1, -1, -1):
            if state[row, col] == 0:
                state[row, col] = player
                break

        return state

    def _check_winner(self, state):
        """
        Check if there's a winner in the current state.

        Args:
            state: Current state of the game

        Returns:
            The winner (1 or 2), 0 for draw, None if game is not over
        """
        # Check horizontal
        for row in range(self.rows):
            for col in range(self.cols - 3):
                if state[row, col] != 0 and state[row, col] == state[row, col+1] == state[row, col+2]
                     == state[row, col+3]:
                    return state[row, col]

        # Check vertical
        for row in range(self.rows - 3):
            for col in range(self.cols):
                if state[row, col] != 0 and state[row, col] == state[row+1, col] == state[row+2, col]
                     == state[row+3, col]:
                    return state[row, col]

        # Check diagonal (positive slope)
        for row in range(self.rows - 3):
            for col in range(self.cols - 3):
                if state[row, col] != 0 and state[row, col] == state[row+1, col+1] == state[row+2,
                     col+2] == state[row+3, col+3]:
                    return state[row, col]

        # Check diagonal (negative slope)
        for row in range(3, self.rows):
            for col in range(self.cols - 3):
                if state[row, col] != 0 and state[row, col] == state[row-1, col+1] == state[row-2,
                     col+2] == state[row-3, col+3]:
                    return state[row, col]

        # Check for draw (top row filled)
        if np.all(state[0, :] != 0):
            return 0  # Draw

        # Game not over yet
        return None
    def _count_immediate_threats(self, state):
        """
        Count the number of immediate winning threats for the current player.
        An immediate threat is an empty position that would result in a win.

        Args:
            state: Current state of the game

        Returns:
            Number of immediate threats
        """
        threats = 0

        # Try each possible move
        for col in range(self.cols):
            if state[0, col] != 0:  # Column is full
                continue

            # Find where piece would land
            for row in range(self.rows-1, -1, -1):
                if state[row, col] == 0:
                    # Try the move
                    test_state = state.copy()
                    test_state[row, col] = self.player

                    # Check if this creates a win
                    if self._is_winning_move(test_state, row, col):
                        threats += 1
```

```
                    break

        return threats

    def _is_winning_move(self, state, row, col):
        """
        Check if the last move at (row, col) creates a win.
        More efficient than checking the entire board.

        Args:
            state: Current state of the game
            row: Row of last move
            col: Column of last move

        Returns:
            True if the move creates a win
        """
        player = state[row, col]

        # Check horizontal
        count = 0
        for c in range(max(0, col-3), min(self.cols, col+4)):
            if state[row, c] == player:
                count += 1
                if count == 4:
                    return True
            else:
                count = 0

        # Check vertical
        count = 0
        for r in range(max(0, row-3), min(self.rows, row+4)):
            if state[r, col] == player:
                count += 1
                if count == 4:
                    return True
            else:
                count = 0

        # Check diagonal (positive slope)
        count = 0
        for i in range(-3, 4):
            r = row + i
            c = col + i
            if 0 <= r < self.rows and 0 <= c < self.cols:
                if state[r, c] == player:
                    count += 1
                    if count == 4:
                        return True
                else:
                    count = 0

        # Check diagonal (negative slope)
        count = 0
        for i in range(-3, 4):
            r = row - i
            c = col + i
            if 0 <= r < self.rows and 0 <= c < self.cols:
                if state[r, c] == player:
                    count += 1
                    if count == 4:
                        return True
                else:
                    count = 0

        return False

    def _evaluate_board(self, state):
        """
        Evaluate the current board state using a heuristic.
        For Connect-4, we'll use a weighted scoring system based on potential connections.

        Args:
            state: Current state of the game

        Returns:
            Numerical score for the board state
        """
        score = 0

        # Check all possible four-in-a-row windows and score them

        # Horizontal windows
        for row in range(self.rows):
            for col in range(self.cols - 3):
                window = state[row, col:col+4]
                score += self._evaluate_window(window)

        # Vertical windows
        for row in range(self.rows - 3):
            for col in range(self.cols):
```

```
                window = state[row:row+4, col]
                score += self._evaluate_window(window)

        # Positive diagonal windows
        for row in range(self.rows - 3):
            for col in range(self.cols - 3):
                window = np.array([state[row+i, col+i] for i in range(4)])
                score += self._evaluate_window(window)

        # Negative diagonal windows
        for row in range(3, self.rows):
            for col in range(self.cols - 3):
                window = np.array([state[row-i, col+i] for i in range(4)])
                score += self._evaluate_window(window)

        # Prefer center columns (better positions strategically)
        center_col = self.cols // 2
        for row in range(self.rows):
            if state[row, center_col] == self.player:
                # More weight to lower positions
                score += 3 * (self.rows - row)

        # Vertical threat bonus (they're harder to block)
        for col in range(self.cols):
            for row in range(self.rows - 3):
                window = state[row:row+4, col]
                our_pieces = np.count_nonzero(window == self.player)
                empty_slots = np.count_nonzero(window == 0)
                if our_pieces == 3 and empty_slots == 1:
                    score += 2  # Additional bonus for vertical threats

        # Detect forced moves (immediate threats)
        immediate_threats = self._count_immediate_threats(state)
        if immediate_threats > 1:
            score += 50  # Multiple threats usually lead to forced win

        return score

    def _evaluate_window(self, window):
        """
        Evaluate a window of 4 slots for potential connections.

        Args:
            window: Array of 4 positions

        Returns:
            Score for this window
        """
        our_pieces = np.count_nonzero(window == self.player)
        opponent_pieces = np.count_nonzero(window == (3 - self.player))
        empty_slots = np.count_nonzero(window == 0)

        if our_pieces == 4:
            return 100  # We win
        elif our_pieces == 3 and empty_slots == 1:
            return 5  # Potential win next move
        elif our_pieces == 2 and empty_slots == 2:
            return 2  # Potential future win

        if opponent_pieces == 3 and empty_slots == 1:
            return -4  # Block opponent win

        return 0

    def _evaluate_terminal(self, winner):
        """
        Evaluate a terminal state.

        Args:
            winner: The winner (1 or 2), 0 for draw

        Returns:
            Numerical score for the terminal state
        """
        if winner == 0:  # Draw
            return 0
        elif winner == self.player:  # We win
            return 1000000  # Very high value
        else:  # Opponent wins
            return -1000000  # Very low value
```

C Q-Learning Implementation

C.1 Q-Learning Base

```
import numpy as np
```

```python
import random
import pickle
import os
from abc import ABC, abstractmethod

class QLearningAgent(ABC):
    def __init__(self, player_number=1, alpha=0.1, gamma=0.9, epsilon=0.1,
                 epsilon_decay=0.999, epsilon_min=0.01, metrics_manager=None):
        """
        Initialize the Q-learning agent.

        Args:
            player_number: The player number (1 or 2)
            alpha: Learning rate
            gamma: Discount factor
            epsilon: Exploration rate
            epsilon_decay: Rate at which epsilon decays
            epsilon_min: Minimum exploration rate
            metrics_manager: Metrics manager for tracking stats
        """
        self.player_number = player_number
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.epsilon_min = epsilon_min
        self.metrics_manager = metrics_manager

        # Initialize Q-table
        self.q_table = {}

        # Set rewards
        self.reward_win = 1.0
        self.reward_loss = -1.0
        self.reward_draw = 0.0
        self.reward_move = -0.01  # Small negative reward for each move to encourage faster winning

        # Training history
        self.training_stats = {
            'episode_rewards': [],
            'win_rate': [],
            'episode_lengths': []
        }

    def decay_epsilon(self):
        """Decay the exploration rate."""
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def get_q_value(self, state, action):
        """Get Q-value for state-action pair. Return 0 if not visited before."""
        state_key = self.state_to_key(state)
        if state_key in self.q_table and action in self.q_table[state_key]:
            return self.q_table[state_key][action]
        return 0.0

    def update_q_value(self, state, action, reward, next_state):
        """Update Q-value for state-action pair."""
        state_key = self.state_to_key(state)
        next_state_key = self.state_to_key(next_state)

        # Initialize q_table entry if it doesn't exist
        if state_key not in self.q_table:
            self.q_table[state_key] = {}

        # Get current Q-value
        current_q = self.get_q_value(state, action)

        # Get max Q-value for next state
        next_max_q = 0.0
        if next_state_key in self.q_table:
            next_q_values = self.q_table[next_state_key].values()
            if next_q_values:
                next_max_q = max(next_q_values)

        # Update rule: Q(s,a) = Q(s,a) + alpha * (r + gamma * max(Q(s',a')) - Q(s,a))
        new_q = current_q + self.alpha * (reward + self.gamma * next_max_q - current_q)
        self.q_table[state_key][action] = new_q

    def choose_action(self, state, legal_moves, training=False):
        """Choose an action using epsilon-greedy policy."""
        if training and random.random() < self.epsilon:
            # Exploration: choose a random action
            return random.choice(legal_moves) if legal_moves else None

        # Exploitation: choose the best action
        best_action = None
        best_value = float('-inf')

        # Shuffle the legal moves to break ties randomly
        random.shuffle(legal_moves)
```

```python
        for action in legal_moves:
            action_value = self.get_q_value(state, action)
            if action_value > best_value:
                best_value = action_value
                best_action = action

        return best_action

    def get_move(self, state):
        """Get a move for the current state (used during gameplay)."""
        legal_moves = self.get_legal_moves(state)
        if not legal_moves:
            return None
        return self.choose_action(state, legal_moves, training=False)

    def train(self, num_episodes=10000, eval_interval=500, eval_games=50, opponent=None):
        """Train the agent through self-play or against an opponent."""
        total_rewards = []

        # Check if metrics manager is available
        if self.metrics_manager:
            self.metrics_manager.set_q_table(self.q_table)

        for episode in range(1, num_episodes + 1):
            # Reset the game
            game = self.create_game()
            state = game.get_state()
            done = False
            episode_reward = 0
            moves = 0

            while not done:
                # Get legal moves
                legal_moves = game.get_legal_moves()
                if not legal_moves:
                    break

                # Choose action
                action = self.choose_action(state, legal_moves, training=True)

                # Make the move
                game.make_move(action)
                next_state = game.get_state()
                moves += 1

                # Check if game is over
                if game.is_game_over():
                    winner = game.get_winner()
                    if winner == self.player_number:
                        reward = self.reward_win
                    elif winner == 0:  # Draw
                        reward = self.reward_draw
                    else:  # Loss
                        reward = self.reward_loss
                    done = True
                else:
                    reward = self.reward_move

                    # If playing against an opponent, let opponent make a move
                    if opponent and not done:
                        opponent_move = opponent.get_move(game.get_state())
                        if opponent_move:
                            game.make_move(opponent_move)
                            # Check if opponent won
                            if game.is_game_over():
                                winner = game.get_winner()
                                if winner == 3 - self.player_number:  # Opponent won
                                    reward = self.reward_loss
                                elif winner == 0:  # Draw
                                    reward = self.reward_draw
                                done = True
                            next_state = game.get_state()
                        else:
                            done = True

                # Update Q-values
                self.update_q_value(state, action, reward, next_state)

                # Update state
                state = next_state
                episode_reward += reward

            # Decay exploration rate
            self.decay_epsilon()

            # Record stats
            total_rewards.append(episode_reward)
            self.training_stats['episode_rewards'].append(episode_reward)
            self.training_stats['episode_lengths'].append(moves)
```

```
            # Log with metrics manager
            if self.metrics_manager:
                self.metrics_manager.record_q_learning_reward(episode, episode_reward)

            # Periodically evaluate the agent
            if episode % eval_interval == 0:
                win_rate = self.evaluate(eval_games, opponent)
                self.training_stats['win_rate'].append((episode, win_rate))

                if self.metrics_manager:
                    self.metrics_manager.print_q_table_memory()

                print(f"Episode {episode}/{num_episodes}: Win rate: {win_rate:.2f}, "
                      f"Epsilon: {self.epsilon:.3f}, "
                      f"Q-table size: {len(self.q_table)}")

        return self.training_stats

    def evaluate(self, num_games=100, opponent=None):
        """Evaluate the agent by playing against an opponent or randomly."""
        results = []

        for _ in range(num_games):
            game = self.create_game()
            done = False

            while not done:
                # Agent's turn
                if game.current_player == self.player_number:
                    action = self.get_move(game.get_state())
                    if action is None:
                        break
                    game.make_move(action)
                # Opponent's turn
                else:
                    if opponent:
                        opp_action = opponent.get_move(game.get_state())
                    else:
                        # Random opponent
                        legal_moves = game.get_legal_moves()
                        opp_action = random.choice(legal_moves) if legal_moves else None

                    if opp_action is None:
                        break
                    game.make_move(opp_action)

                # Check if game is over
                if game.is_game_over():
                    winner = game.get_winner()
                    if winner == self.player_number:
                        results.append('win')
                    elif winner == 0:
                        results.append('draw')
                    else:
                        results.append('loss')
                    done = True

        # Record evaluation results with metrics manager
        if self.metrics_manager:
            self.metrics_manager.record_q_learning_evaluation(len(self.training_stats['episode_rewards']),
                results)

        win_rate = results.count('win') / len(results) if results else 0
        return win_rate

    def save(self, filepath):
        """Save the Q-table to a file."""
        with open(filepath, 'wb') as f:
            pickle.dump(self.q_table, f)

        # Also save training stats
        stats_filepath = f"{os.path.splitext(filepath)[0]}_stats.pkl"
        with open(stats_filepath, 'wb') as f:
            pickle.dump(self.training_stats, f)

    def load(self, filepath):
        """Load the Q-table from a file."""
        with open(filepath, 'rb') as f:
            self.q_table = pickle.load(f)

        # Also try to load training stats
        stats_filepath = f"{os.path.splitext(filepath)[0]}_stats.pkl"
        if os.path.exists(stats_filepath):
            with open(stats_filepath, 'rb') as f:
                self.training_stats = pickle.load(f)

        # Update metrics manager
        if self.metrics_manager:
            self.metrics_manager.set_q_table(self.q_table)

    @abstractmethod
```

```
    def state_to_key(self, state):
        """Convert state to a key that can be used in the Q-table.
        Must be implemented by subclasses."""
        pass

    @abstractmethod
    def get_legal_moves(self, state):
        """Get legal moves for the given state.
        Must be implemented by subclasses."""
        pass

    @abstractmethod
    def create_game(self):
        """Create a new game instance.
        Must be implemented by subclasses."""
        pass
```

C.2 Q-Learning Tic Tac Toe

```
    import numpy as np
from games.tic_tac_toe import TicTacToe
from .qlearning_base import QLearningAgent

class QLearningTicTacToe(QLearningAgent):
    def __init__(self, player_number=1, alpha=0.3, gamma=0.9, epsilon=0.3,
                 epsilon_decay=0.9999, epsilon_min=0.01, metrics_manager=None):
        """
        Initialize the Q-learning agent for Tic-Tac-Toe.

        Args:
            player_number: The player number (1 or 2)
            alpha: Learning rate
            gamma: Discount factor
            epsilon: Exploration rate
            epsilon_decay: Rate at which epsilon decays
            epsilon_min: Minimum exploration rate
            metrics_manager: Metrics manager for tracking stats
        """
        super().__init__(player_number, alpha, gamma, epsilon,
                         epsilon_decay, epsilon_min, metrics_manager)

        # Higher rewards for Tic-Tac-Toe due to shorter game length
        self.reward_win = 1.0
        self.reward_loss = -1.0
        self.reward_draw = 0.2  # Draws are better than losses
        self.reward_move = -0.05  # Small penalty for each move

    def state_to_key(self, state):
        """
        Convert the game state to a hashable key for the Q-table.

        Args:
            state: 3x3 numpy array representing the game board

        Returns:
            tuple: A hashable representation of the state
        """
        # Convert the board state to a tuple of tuples
        # This ensures the state is hashable and can be used as a dictionary key
        return tuple(map(tuple, state))

    def get_legal_moves(self, state):
        """
        Get all legal moves for the current state.

        Args:
            state: The game state

        Returns:
            list: List of legal moves as (row, col) tuples
        """
        return [(i, j) for i in range(3) for j in range(3) if state[i, j] == 0]

    def create_game(self):
        """Create a new Tic-Tac-Toe game instance."""
        game = TicTacToe()
        # If player 2, make a random first move as player 1
        if self.player_number == 2:
            # Choose a random move for player 1
            moves = self.get_legal_moves(game.get_state())
            if moves:
                move = moves[np.random.randint(len(moves))]
                game.make_move(move)
        return game

    def get_symmetries(self, state, action):
        """
        Get all symmetric states and corresponding actions.
```

```python
        This helps the agent learn faster by exploiting the symmetry of the game.

        Args:
            state: The game state
            action: The action taken in that state

        Returns:
            list: List of (state_key, action) pairs for all symmetries
        """
        state_array = np.array(state).reshape(3, 3)
        row, col = action

        symmetries = []

        # Original
        symmetries.append((tuple(map(tuple, state_array)), (row, col)))

        # Rotate 90 degrees
        rot90 = np.rot90(state_array)
        new_row, new_col = 2 - col, row
        symmetries.append((tuple(map(tuple, rot90)), (new_row, new_col)))

        # Rotate 180 degrees
        rot180 = np.rot90(rot90)
        new_row, new_col = 2 - row, 2 - col
        symmetries.append((tuple(map(tuple, rot180)), (new_row, new_col)))

        # Rotate 270 degrees
        rot270 = np.rot90(rot180)
        new_row, new_col = col, 2 - row
        symmetries.append((tuple(map(tuple, rot270)), (new_row, new_col)))

        # Flip horizontally
        flip_h = np.fliplr(state_array)
        new_row, new_col = row, 2 - col
        symmetries.append((tuple(map(tuple, flip_h)), (new_row, new_col)))

        # Flip vertically
        flip_v = np.flipud(state_array)
        new_row, new_col = 2 - row, col
        symmetries.append((tuple(map(tuple, flip_v)), (new_row, new_col)))

        # Flip along main diagonal
        flip_diag = np.transpose(state_array)
        new_row, new_col = col, row
        symmetries.append((tuple(map(tuple, flip_diag)), (new_row, new_col)))

        # Flip along other diagonal
        flip_diag2 = np.rot90(np.transpose(rot90))
        new_row, new_col = 2 - col, 2 - row
        symmetries.append((tuple(map(tuple, flip_diag2)), (new_row, new_col)))

        return symmetries

    def update_q_value(self, state, action, reward, next_state):
        """
        Update Q-value for state-action pair and all its symmetries.

        Args:
            state: The game state
            action: The action taken
            reward: The reward received
            next_state: The resulting state
        """
        # Get all symmetric states and actions
        symmetries = self.get_symmetries(state, action)

        for sym_state, sym_action in symmetries:
            state_key = sym_state

            # Initialize q_table entry if it doesn't exist
            if state_key not in self.q_table:
                self.q_table[state_key] = {}

            # Get current Q-value
            if sym_action in self.q_table[state_key]:
                current_q = self.q_table[state_key][sym_action]
            else:
                current_q = 0.0

            # Get max Q-value for next state
            next_state_key = self.state_to_key(next_state)
            next_max_q = 0.0
            if next_state_key in self.q_table:
                next_q_values = self.q_table[next_state_key].values()
                if next_q_values:
                    next_max_q = max(next_q_values)

            # Update rule: Q(s,a) = Q(s,a) + alpha * (r + gamma * max(Q(s',a')) - Q(s,a))
            new_q = current_q + self.alpha * (reward + self.gamma * next_max_q - current_q)
            self.q_table[state_key][sym_action] = new_q
```

C.3 Q-Learning Connect 4

```
    import random
import numpy as np
from games.connect4 import Connect4
from .qlearning_base import QLearningAgent

class QLearningConnect4(QLearningAgent):
    def __init__(self, player_number=1, alpha=0.2, gamma=0.95, epsilon=0.3,
                 epsilon_decay=0.9995, epsilon_min=0.01, metrics_manager=None):
        """
        Initialize the Q-learning agent for Connect-4.

        Args:
            player_number: The player number (1 or 2)
            alpha: Learning rate
            gamma: Discount factor
            epsilon: Exploration rate
            epsilon_decay: Rate at which epsilon decays
            epsilon_min: Minimum exploration rate
            metrics_manager: Metrics manager for tracking stats
        """
        super().__init__(player_number, alpha, gamma, epsilon,
                         epsilon_decay, epsilon_min, metrics_manager)

        # Adjust rewards for Connect-4
        self.reward_win = 1.0
        self.reward_loss = -1.0
        self.reward_draw = 0.0
        self.reward_move = -0.01  # Small penalty for each move

        # Caching for move detection
        self._horizontal_window_indices = self._precompute_horizontal_windows()
        self._vertical_window_indices = self._precompute_vertical_windows()
        self._diagonal_window_indices = self._precompute_diagonal_windows()

    def state_to_key(self, state):
        """
        Convert the game state to a hashable key for the Q-table.
        For Connect-4, we use a tuple of tuples representation.

        Args:
            state: 6x7 numpy array representing the game board

        Returns:
            tuple: A hashable representation of the state
        """
        return tuple(map(tuple, state))

    def get_legal_moves(self, state):
        """
        Get all legal moves for the current state in Connect-4.
        Legal moves are columns where a piece can be dropped.

        Args:
            state: The game state

        Returns:
            list: List of legal moves (column indices)
        """
        return [col for col in range(7) if state[0][col] == 0]

    def create_game(self):
        """Create a new Connect-4 game instance."""
        game = Connect4()
        # If player 2, make a random first move as player 1
        if self.player_number == 2:
            # Choose a random move for player 1
            moves = self.get_legal_moves(game.get_state())
            if moves:
                move = moves[np.random.randint(len(moves))]
                game.make_move(move)
        return game

    def _precompute_horizontal_windows(self):
        """Precompute all possible horizontal 4-in-a-row window indices."""
        windows = []
        for row in range(6):
            for col in range(4):
                window = [(row, col + i) for i in range(4)]
                windows.append(window)
        return windows

    def _precompute_vertical_windows(self):
        """Precompute all possible vertical 4-in-a-row window indices."""
        windows = []
```

```
        for row in range(3):
            for col in range(7):
                window = [(row + i, col) for i in range(4)]
                windows.append(window)
        return windows

    def _precompute_diagonal_windows(self):
        """Precompute all possible diagonal 4-in-a-row window indices."""
        windows = []
        # Positive slope diagonals
        for row in range(3, 6):
            for col in range(4):
                window = [(row - i, col + i) for i in range(4)]
                windows.append(window)
        # Negative slope diagonals
        for row in range(3):
            for col in range(4):
                window = [(row + i, col + i) for i in range(4)]
                windows.append(window)
        return windows

    def _count_window(self, state, window, player):
        """
        Count pieces in a window for a given player.
        Returns the count of player's pieces if the window doesn't contain opponent pieces,
        otherwise returns 0.
        """
        count = 0
        opponent = 3 - player
        for row, col in window:
            if state[row][col] == opponent:
                return 0
            if state[row][col] == player:
                count += 1
        return count

    def _detect_threats(self, state, player):
        """
        Detect immediate threats (3-in-a-row with an empty space) for the given player.

        Args:
            state: The game state
            player: The player to check threats for

        Returns:
            list: List of column indices where there are immediate threats
        """
        threats = []
        opponent = 3 - player

        # Check horizontal threats
        for window in self._horizontal_window_indices:
            # Count player and empty spaces in window
            player_count = 0
            empty_pos = None

            for row, col in window:
                if state[row][col] == player:
                    player_count += 1
                elif state[row][col] == 0:
                    empty_pos = (row, col)

            # If 3 player pieces and 1 empty, it's a threat
            if player_count == 3 and empty_pos:
                # Make sure the empty position is valid (either at bottom or has support below)
                empty_row, empty_col = empty_pos
                if empty_row == 5 or state[empty_row + 1][empty_col] != 0:
                    if empty_col not in threats:
                        threats.append(empty_col)

        # Check vertical threats
        for window in self._vertical_window_indices:
            # For vertical windows, we need 3 player pieces and the top is empty
            cells = [(r, c) for r, c in window]
            player_count = sum(1 for r, c in cells if state[r][c] == player)
            bottom_cell = max(cells, key=lambda x: x[0])  # Cell with largest row index
            top_cell = min(cells, key=lambda x: x[0])     # Cell with smallest row index

            if player_count == 3 and state[top_cell[0]][top_cell[1]] == 0:
                if top_cell[1] not in threats:
                    threats.append(top_cell[1])

        # Check diagonal threats (both directions)
        for window in self._diagonal_window_indices:
            player_count = 0
            empty_pos = None

            for row, col in window:
                if state[row][col] == player:
                    player_count += 1
                elif state[row][col] == 0:
```

```
                empty_pos = (row, col)

        # If 3 player pieces and 1 empty, check if it's a valid move
        if player_count == 3 and empty_pos:
            empty_row, empty_col = empty_pos
            # Check if the move is valid (either at bottom or has support)
            if empty_row == 5 or state[empty_row + 1][empty_col] != 0:
                if empty_col not in threats:
                    threats.append(empty_col)

    return threats

def _detect_double_threats(self, state, player):
    """
    Detect positions that would create multiple threats simultaneously.
    These are usually winning moves.

    Args:
        state: The game state
        player: The player to check threats for

    Returns:
        list: List of column indices that create multiple threats
    """
    double_threats = []

    # For each legal move, simulate it and count resulting threats
    for col in range(7):
        # Skip if column is full
        if state[0][col] != 0:
            continue

        # Find the row where the piece would land
        for row in range(5, -1, -1):
            if state[row][col] == 0:
                # Simulate the move
                temp_state = [list(row) for row in state]
                temp_state[row][col] = player

                # Count threats after this move
                threats = self._detect_threats(temp_state, player)
                if len(threats) >= 2:
                    double_threats.append(col)
                break

    return double_threats

def get_heuristic_features(self, state):
    """
    Extract heuristic features from the state to augment the Q-learning.
    Enhanced with strategies from minimax evaluation function.

    Features returned:
    - Number of potential winning lines with 1, 2, or 3 pieces
    - Center column control with positional weighting
    - Vertical threat recognition
    - Multiple threats detection

    Args:
        state: The game state

    Returns:
        dict: Dictionary of features
    """
    features = {
        'one_piece': 0,
        'two_pieces': 0,
        'three_pieces': 0,
        'center_control': 0,
        'vertical_threats': 0,
        'has_immediate_threat': 0,
        'has_double_threat': 0,
        'blocking_opponent_win': 0
    }

    player = self.player_number
    opponent = 3 - player

    # Count pieces in horizontal windows
    for window in self._horizontal_window_indices:
        count = self._count_window(state, window, player)
        if count == 1:
            features['one_piece'] += 1
        elif count == 2:
            features['two_pieces'] += 1
        elif count == 3:
            features['three_pieces'] += 1
            # Check if this is a valid threat (can be played immediately)
            for row, col in window:
                if state[row][col] == 0:
                    # If this empty cell is at bottom or has support
```

```python
                    if row == 5 or state[row+1][col] != 0:
                        features['has_immediate_threat'] = 1

        # Count pieces in vertical windows
        for window in self._vertical_window_indices:
            count = self._count_window(state, window, player)
            if count == 1:
                features['one_piece'] += 1
            elif count == 2:
                features['two_pieces'] += 1
            elif count == 3:
                features['three_pieces'] += 1
                features['vertical_threats'] += 1
                features['has_immediate_threat'] = 1

        # Count pieces in diagonal windows
        for window in self._diagonal_window_indices:
            count = self._count_window(state, window, player)
            if count == 1:
                features['one_piece'] += 1
            elif count == 2:
                features['two_pieces'] += 1
            elif count == 3:
                features['three_pieces'] += 1
                # Check if this is a valid threat (can be played immediately)
                for row, col in window:
                    if state[row][col] == 0:
                        # If this empty cell is at bottom or has support
                        if row == 5 or state[row+1][col] != 0:
                            features['has_immediate_threat'] = 1

        # Center column control with positional weighting
        center_col = 3
        for row in range(6):
            if state[row][center_col] == player:
                # More weight to lower positions (like in minimax)
                features['center_control'] += (6 - row)

        # Check for multiple threats
        threats = self._detect_threats(state, player)
        if len(threats) >= 2:
            features['has_double_threat'] = 1

        # Check if we need to block opponent's immediate win
        opponent_threats = self._detect_threats(state, opponent)
        if opponent_threats:
            features['blocking_opponent_win'] = 1

        return features

    def choose_action(self, state, legal_moves, training=False):
        """
        Choose an action using epsilon-greedy policy combined with enhanced heuristic knowledge.
        This implementation gives a sophisticated evaluation of potential moves.

        Args:
            state: The game state
            legal_moves: List of legal moves
            training: Whether we're in training mode (exploration enabled)

        Returns:
            int: Column to drop the piece
        """
        if not legal_moves:
            return None

        # During training, use epsilon-greedy exploration
        if training and random.random() < self.epsilon:
            return random.choice(legal_moves)

        # Check if we have an immediate winning move
        player_threats = self._detect_threats(state, self.player_number)
        if player_threats:
            # Prioritize the winning move
            return player_threats[0]

        # Check if we need to block opponent's immediate win
        opponent = 3 - self.player_number
        opponent_threats = self._detect_threats(state, opponent)
        if opponent_threats:
            # Need to block
            return opponent_threats[0]

        # Check for moves that create multiple threats (usually winning)
        double_threats = self._detect_double_threats(state, self.player_number)
        if double_threats:
            return double_threats[0]

        # Exploitation: choose the best action based on Q-values and enhanced heuristics
        best_action = None
        best_value = float('-inf')
```

```
        # Weight of heuristic values vs Q-values
        heuristic_weight = 0.3

        for action in legal_moves:
            # Get Q-value
            q_value = self.get_q_value(state, action)

            # Simulate the move to get heuristic value
            temp_state = np.array(state)
            for row in range(5, -1, -1):
                if temp_state[row][action] == 0:
                    temp_state[row][action] = self.player_number
                    break

            temp_features = self.get_heuristic_features(temp_state)

            # Calculate heuristic value with improved weights
            heuristic_value = (
                0.1 * temp_features['one_piece'] +
                0.3 * temp_features['two_pieces'] +
                0.8 * temp_features['three_pieces'] +
                0.4 * temp_features['center_control'] +
                0.5 * temp_features['vertical_threats'] +
                2.0 * temp_features['has_immediate_threat'] +
                3.0 * temp_features['has_double_threat'] +
                1.0 * temp_features['blocking_opponent_win']
            )

            # Combine Q-value and heuristic
            total_value = (1 - heuristic_weight) * q_value + heuristic_weight * heuristic_value

            if total_value > best_value:
                best_value = total_value
                best_action = action

        return best_action

    def update_q_value(self, state, action, reward, next_state):
        """
        Update Q-value for state-action pair.
        Connect-4 has too many states for symmetry-based optimization to be practical.

        Args:
            state: The game state
            action: The action taken
            reward: The reward received
            next_state: The resulting state
        """
        state_key = self.state_to_key(state)
        next_state_key = self.state_to_key(next_state)

        # Initialize q_table entry if it doesn't exist
        if state_key not in self.q_table:
            self.q_table[state_key] = {}

        # Get current Q-value
        if action in self.q_table[state_key]:
            current_q = self.q_table[state_key][action]
        else:
            current_q = 0.0

        # Get max Q-value for next state
        next_max_q = 0.0
        if next_state_key in self.q_table:
            next_q_values = self.q_table[next_state_key].values()
            if next_q_values:
                next_max_q = max(next_q_values)

        # Update rule: Q(s,a) = Q(s,a) + alpha * (r + gamma * max(Q(s',a')) - Q(s,a))
        new_q = current_q + self.alpha * (reward + self.gamma * next_max_q - current_q)
        self.q_table[state_key][action] = new_q
```

D Default Opponent Implementation

D.1 Default Opponent Tic Tac Toe

```
import random
import numpy as np

from games.tic_tac_toe import TicTacToe

class DefaultOpponentTTT:
    """A semi-intelligent agent for Tic Tac Toe that:
    - Plays a winning move if available
    - Blocks opponent's winning move if possible
```

```
        - Otherwise plays randomly
        """
        def __init__(self, player_number=2):
            """Initialize the agent.

            Args:
                player_number: 1 for X, 2 for O (default is 2)
            """
            self.player_number = player_number

        def get_move(self, state):
            """Return the next move based on the current state.

            Args:
                state: Current game board state as numpy array

            Returns:
                Move as (row, col) tuple
            """
            game = TicTacToe()
            game.board = state.copy()
            game.current_player = self.player_number

            # Get all legal moves
            legal_moves = game.get_legal_moves()
            if not legal_moves:
                return None

            # Check for winning move
            for move in legal_moves:
                test_game = TicTacToe()
                test_game.board = state.copy()
                test_game.current_player = self.player_number
                test_game.make_move(move)
                if test_game.is_game_over() and test_game.get_winner() == self.player_number:
                    return move

            # Check for blocking opponent's winning move
            opponent = 3 - self.player_number
            for move in legal_moves:
                test_board = state.copy()
                test_board[move[0], move[1]] = opponent

                # Check if this move would make opponent win
                test_game = TicTacToe()
                test_game.board = test_board
                test_game._check_game_over()
                if test_game.is_game_over() and test_game.get_winner() == opponent:
                    return move

            # If middle square is available, take it (strategic advantage)
            if (1, 1) in legal_moves:
                return (1, 1)

            # Prefer corners over sides (strategic advantage)
            corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
            available_corners = [move for move in corners if move in legal_moves]
            if available_corners:
                return random.choice(available_corners)

            # Otherwise, play randomly
            return random.choice(legal_moves)


if __name__ == "__main__":
    # Example of using the default opponent
    from games.tic_tac_toe import TicTacToeUI, PlayerType, GameMode

    # Create the game UI
    game_ui = TicTacToeUI()

    # Create the default opponent and set as player 2
    default_opponent = DefaultOpponentTTT(player_number=2)
    game_ui.set_player2_agent(default_opponent)

    # Set the game mode to Human vs Semi-Intelligent
    game_ui.game_mode = GameMode.HUMAN_VS_SEMI
    game_ui.player1_type = PlayerType.HUMAN
    game_ui.player2_type = PlayerType.SEMI_INTELLIGENT

    # Run the game
    game_ui.run()
```

D.2 Default Opponent Connect 4

```
import random
import numpy as np
```

```python
from games.connect4 import Connect4

class DefaultOpponentC4:
    """A semi-intelligent agent for Connect 4 that:
    - Plays a winning move if available
    - Blocks opponent's winning move if possible
    - Prefers center columns (strategic advantage)
    - Otherwise plays randomly
    """
    def __init__(self, player_number=2):
        """Initialize the agent.

        Args:
            player_number: 1 for Red, 2 for Yellow (default is 2)
        """
        self.player_number = player_number

    def get_move(self, state):
        """Return the next move based on the current state.

        Args:
            state: Current game board state as numpy array

        Returns:
            Move as column index (0-6)
        """
        game = Connect4()
        game.board = state.copy()
        game.current_player = self.player_number

        # Get all legal moves
        legal_moves = game.get_legal_moves()
        if not legal_moves:
            return None

        # Check for winning move
        for col in legal_moves:
            test_game = Connect4()
            test_game.board = state.copy()
            test_game.current_player = self.player_number
            test_game.make_move(col)
            if test_game.is_game_over() and test_game.get_winner() == self.player_number:
                return col

        # Check for blocking opponent's winning move
        opponent = 3 - self.player_number
        for col in legal_moves:
            test_game = Connect4()
            test_game.board = state.copy()
            test_game.current_player = opponent
            test_game.make_move(col)
            if test_game.is_game_over() and test_game.get_winner() == opponent:
                return col

        # Prefer middle columns
        # The closer to the middle, the higher the probability of being chosen
        weights = []
        middle = 3  # For a 7-column board, the middle is index 3
        for col in legal_moves:
            # Calculate weight based on distance from middle
            distance = abs(col - middle)
            weight = 7 - distance  # Higher weight for columns closer to middle
            weights.append(weight)

        # Normalize weights to probabilities
        total_weight = sum(weights)
        probabilities = [w / total_weight for w in weights]

        # Choose column based on weights
        return random.choices(legal_moves, weights=probabilities, k=1)[0]


if __name__ == "__main__":
    # Example of using the default opponent
    from games.connect4 import Connect4UI, PlayerType, GameMode

    # Create the game UI
    game_ui = Connect4UI()

    # Create the default opponent and set as player 2
    default_opponent = DefaultOpponentC4(player_number=2)
    game_ui.set_player2_agent(default_opponent)

    # Set the game mode to Human vs Semi-Intelligent
    game_ui.game_mode = GameMode.HUMAN_VS_SEMI
    game_ui.player1_type = PlayerType.HUMAN
    game_ui.player2_type = PlayerType.SEMI_INTELLIGENT

    # Run the game
    game_ui.run()
```