Vanishing TicTacToe

Wael Ben Slima
National Taiwan University
a13922118@ntu.edu.tw

Alexa Gomez
National Taiwan University
b09902083@ntu.edu.tw

Eric Bennett
National Taiwan University
t13902169@ntu.edu.tw

Hassan Rizvi
National Taiwan University
t13902181@ntu.edu.tw

Kai Li
National Taiwan University
a13922122@ntu.edu.tw

Abstract

Vanishing Tic Tac Toe presents an intriguing challenge in the domain of game-playing AI due to its deceptively simple rules and compact state space, which nonetheless give rise to rich strategic complexity. In this project, we investigate the application of Deep Reinforcement Learning (DRL) techniques to this niche game, aiming to develop an agent that achieves superhuman performance. Despite the limited number of possible game states, the dynamic nature of Vanishing Tic Tac Toe demands nuanced decision-making and long-term planning, making it a compelling testbed for evaluating the effectiveness of DRL methods in low-dimensional yet nontrivial environments. Our approach involves training agents using state-of-the-art deep learning architectures and reinforcement learning algorithms, with a focus on sample efficiency and convergence behavior. Ideally, the resulting agent will demonstrate strategic mastery surpassing that of human players, offering insights into how deep learning can uncover optimal policies in constrained yet complex game environments.

1 Introduction

Tic Tac Toe is widely recognized as one of the simplest deterministic games, characterized by a 3×3 grid and alternating turns where players place a single mark on the board. Its limited state space and easily computable optimal strategies make it a poor candidate for testing advanced machine learning methods. However, by introducing a minimal yet impactful modification—removing a player's oldest piece after their third move—the game transforms dramatically. This variant, known as Vanishing Tic Tac Toe, retains the intuitive mechanics of the original while introducing temporal dynamics and piece lifecycle constraints that add substantial depth to gameplay.

This simple rule change increases the strategic demands of the game by introducing a layer of long-term planning: players must not only focus on forming immediate threats but also account for the aging and eventual disappearance of their pieces. The result is a richer game environment with complex positional play and evolving board control, despite the tiny state space.

The primary motivation of this project is to explore how Deep Reinforcement Learning (DRL) methods perform in such a constrained yet nontrivial setting. By doing so, we aim to better understand the capabilities of DRL in discovering sophisticated strategies within small but strategically rich environments. Vanishing Tic Tac Toe provides a rare opportunity to isolate the learning capabilities of DRL algorithms from the confounding effects of large or noisy state spaces, offering clear insights into their decision-making behaviors and limitations.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

2 Related Work

Earlier works like Widyantoro and Vembrina (2009) used model-free reinforcement learning—specifically the Q-learning algorithm Watkins and Dayan (1992)—to play Tic-Tac-Toe, but they only managed win or tie rates below 50%. Q-learning works by updating Q-values using the Bellman equation and picking actions that maximize the expected future reward. Later, Ho et al. (2023) improved the win/tie rate to about 90%, still using Q-learning with the Bellman update. It's worth noting that Widyantoro and Vembrina (2009) tested their agent against humans, while Ho et al. (2023) used their own programmed opponent. Most research so far has focused on regular Tic-Tac-Toe, but as far as we know, we're the first to apply Deep Reinforcement Learning to Vanishing Tic-Tac-Toe—a harder version where a player's oldest piece disappears after their third move. This rule adds more complexity and increases the state space, making standard model-free methods less effective. Because of that, we use a more capable model based on Double Q-learning van Hasselt et al. (2015), which helps reduce overestimation by separating action selection from value estimation, using two different networks.

3 Environment

We propose the Vanishing Tic Tac Toe environment, a modified version of classical Tic Tac Toe which introduces a temporal decay mechanism: on an $n \times n$ board, after n turns, the oldest of each players' moves begin to disappear each turn. For the purposes of this report we focused on the traditional 3x3 board size, with decay beginning after move 3, but the environment supports play for any size board.

The decay mechanic forces agents to reason both about the current board, and the temporal ordering of previous actions, since pieces begin to be removed after the first n moves. Unlike traditional Tic Tac Toe, which always terminates in at most five moves per player, Vanishing Tic Tac Toe allows for potentially limitless gameplay, with matches extending over hundreds or thousands of turns. As a result, traditional memorization or greedy-based strategies become inadequate, and success requires long-term planning and sequencing.

Figure 1 illustrates the details of the environment at any step after the 3rd move.

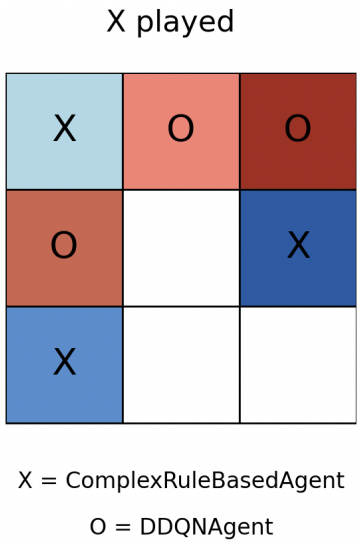

Figure 1: An example board configuration in Vanishing Tic Tac Toe. In order to be more easily analyzable by humans, the older pieces (closer to disappearing) are given a darker shade of their respective colors than the lighter ones.

By introducing decaying pieces, this environment encourages policies that reason over temporal dynamics, making it an ideal testbed for evaluating agents designed for environments with delayed consequences and partial observability.

3.1 Implementation with OpenAI Gym

The environment is implemented as a custom subclass of gym.Env under the class VanishingTicTacToeEnv. It conforms to the Gym API by implementing the reset(), step(), and render() methods. The board is represented as a flattened n × n NumPy array, and move history is stored for each player using a deque structure to enable FIFO removal of old moves.

Each time a player makes a move, if their history has reached the defined disappearturn threshold, the oldest move is removed and its corresponding cell on the board is cleared. This mechanic creates an ever-changing tactical landscape, forcing players to balance aggression with persistence

3.2 State and Observation Space

The environment's observation space is a structured dictionary composed of:

- `board`: A flattened array representing the current board state (-1 for `O`, 1 for `X`, 0 for empty)
- `history_x`: A fixed-length queue of previous move positions made by player `X`
- `history_o`: A fixed-length queue of previous move positions made by player `O`
- `current_player`: The player whose turn it is (1 for `X`, -1 for `O`)

3.3 Action Space and Legal Moves

The action space is `Discrete`(n^2), where each index corresponds to a cell in the $n \times n$ grid. The environment checks for action legality internally: if an agent attempts to place a mark in an already occupied cell, the game terminates and is marked as invalid.

3.4 Reward Structure

The reward function is sparse and symmetric:

- $+1.0$ if the current player wins
- -1.0 if the current player loses
- 0.0 if the game continues

Invalid moves result in immediate termination and no reward.

This formulation encourages long-term planning to create win conditions before a player's own moves vanish.

3.5 Core Game Logic

`step(action)` Processes the agent's move, applies vanishing mechanics, checks for a winner, updates the player turn, and returns a new observation and reward.

```
if len(history) >= self.disappear_turn:
    oldest_pos = history.popleft()
    self.board[oldest_pos] = 0

self.board[action] = self.current_player
history.append(action)
```

`check_winner()` Iterates over rows, columns, and diagonals to check if the current player has aligned n marks.

`render()` Displays the board in the terminal using ASCII characters:

```
 X | O | O
```

```
---+---+---
 O |   | X
---+---+---
 X   |   |
```

4 Rule-Based Agents

4.1 Random Agent

The `RandomAgent` selects legal actions uniformly at random. It serves as a baseline for comparison and lacks strategic depth.

```
def act(self, observation):
    board = observation["board"]
    legal = [i for i, v in enumerate(board) if v == 0]
    return np.random.choice(legal)
```

4.2 SimpleRuleBasedAgent

This agent implements two basic strategies:

- Win immediately if possible
- Otherwise, place a mark adjacent to a previously placed mark

```
# Prefer adjacent moves if no win found
adjacent_candidates = set()
for r, c in own_move_coords:
    for dr, dc in neighborhood_offsets:
        nr, nc = r + dr, c + dc
        if valid_cell(nr, nc):
            idx = to_flat_index(nr, nc)
            if board[idx] == 0:
                adjacent_candidates.add(idx)
```

Fallback is to select any legal move if no adjacency is found. This agent captures local tactical awareness but lacks anticipation.

4.2.1 Moderate Rule-Based Agent

This agent adds basic lookahead:

- If a winning move exists, play it.
- If the opponent has a winning move, block it.
- Otherwise, play randomly.

```
for move in legal_actions:
    temp_board[move] = player
    if self.check_win(temp_board, size, player):
        return move

for move in legal_actions:
    temp_board[move] = opponent
    if self.check_win(temp_board, size, opponent):
        return move
```

4.3 ComplexRuleBasedAgent

This agent implements full-fledged game analysis with six strategic layers:

1. Win immediately
2. Block opponent win
3. Create fork (two or more simultaneous winning paths)
4. Block opponent fork
5. Positional preference (center $>$ corners $>$ edges)
6. Safety check (avoid giving opponent an immediate win)

Example: Fork Detection

```
def _creates_fork(self, board, hist, pos, player):
    b2, h2 = self._simulate(board, hist, pos, player)
    return self._count_wins(b2, h2, player) >= 2
```

Safe Move Validation

```
def _is_safe_move(self, board, hist_me, hist_opp, pos, me, opp):
    b2, _ = self._simulate(board, hist_me, pos, me)
    return not self._opponent_can_win_next(b2, hist_opp, opp)
```

5 Learning Agent: Double DQN

To tackle the strategic complexity of Vanishing Tic Tac Toe, we implemented a Double Deep Q-Network (Double DQN) agent. This approach mitigates the overestimation bias commonly found in standard DQN by decoupling action selection from evaluation. We adopted this method due to its sample efficiency and stability, which are crucial in an environment with sparse rewards and long episodes.

5.1 Architecture and Hyperparameters

The agent's Q-network is a fully connected feed-forward neural network with the following hidden layer dimensions: [528, 256, 128, 256, 528]. Each layer is followed by a ReLU activation, and the final output is a linear layer producing Q-values for each action. Both the policy and target networks share this architecture.

We used the following training hyperparameters:

- Learning rate: 2.5×10^{-4} (Adam optimizer)
- Discount factor $\gamma = 0.99$
- Batch size: 128
- Replay buffer size: 10,000
- Target network update frequency: every 10,000 steps
- Epsilon-greedy exploration: exponentially decayed from 1.0 to 0.05 over 800,000 episodes
- Gradient clipping: value clip at 1.0 and norm clip at 10.0
- N-step returns: $n = 1$

5.2 Training Procedure

The agent was trained for 800,000 episodes. In each episode, it played either as `X` or `O`, alternating turn order accordingly. The opponent for each episode was sampled probabilistically from a pool of rule-based agents:

- 70% chance: ComplexRuleBasedAgent
- 15% chance: ModerateRuleBasedAgent

- 10% chance: SimpleRuleBasedAgent
- 5% chance: RandomAgent

This curriculum balances strong adversarial training with exposure to varied playstyles, helping the agent generalize and avoid overfitting to a single opponent's strategy. Each transition stored in the replay buffer spans one full agent move and one opponent response, capturing the two-turn dynamics necessary for learning long-term consequences.

5.3 Evaluation Results

Our best results came 50,000 episodes into training, where we performed evaluation across a suite of agents, measuring win rates as both first and second player. The agent was evaluated in greedy mode without exploration noise.

Opponent	Win Rate (X)	Win Rate (O)
ComplexRuleBasedAgent	85.9% (9 draws)	95.6% (0 draws)
ModerateRuleBasedAgent	68.7% (20 draws)	49.0% (1 draw)
SimpleRuleBasedAgent	80.0% (10 draws)	58.8% (0 draws)
RandomAgent	80.4% (0 draws)	81.2% (0 draws)

Table 1: Evaluation win rates after 50,000 episodes.

The agent achieves a total average win rate of 74.9% across all matchups, demonstrating robust performance and consistent superiority over all baseline agents. Notably, it performs extremely well against the ComplexRuleBasedAgent, indicating it has developed high-level strategic reasoning that exceeds even handcrafted heuristics.

6 Conclusion and Future Work

In this project, we explored the application of Deep Reinforcement Learning to the novel and strategically rich environment of Vanishing Tic Tac Toe. By leveraging a Double DQN agent with a deep feed-forward architecture and a curriculum of progressively stronger rule-based opponents, we were able to train a policy that significantly outperforms all baseline agents—including handcrafted heuristic strategies.

Our results demonstrate that even in environments with a small state space, introducing temporal dynamics like piece disappearance can yield substantial strategic depth that challenges modern RL algorithms. The agent achieved a 74.9% average win rate across a diverse set of opponents, exhibiting strong long-term planning and adaptability.

For future work, our objective is to investigate the impact of advanced techniques such as prioritized experience replay, dueling architectures, and reward shaping to further accelerate learning. Additionally, scaling the environment to larger board sizes or incorporating stochastic elements could provide richer benchmarks for evaluating generalization and robustness in constrained DRL settings.

7 Code Availability

The codebase, including training scripts, is available at `https://github.com/softly-undefined/tictactoe`.

References

Ho, J., Huang, J., Chang, B., Liu, A., and Liu, Z. (2023). Reinforcement learning: Playing tic-tac-toe. *Authorea Preprints*.

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8:279–292.

Widyantoro, D. H. and Vembrina, Y. G. (2009). Learning to play tic-tac-toe. In *2009 International Conference on Electrical Engineering and Informatics*, volume 01, pages 276–280.