Reinforcement Learning - Tic Tac Toe

Andrew Lau
MEng Robotics and Artificial Intelligence
University College London
andrew.lau.24@ucl.ac.uk

Thun Sahacharoen
MEng Robotics and Artificial Intelligence
University College London
thun.sahacharoen.24@ucl.ac.uk

Sofiya Flenova
MEng Robotics and Artificial Intelligence
University College London
sofiya.flenova.24@ucl.ac.uk

Ted Tang
MEng Robotics and Artificial Intelligence
University College London
ted.tang.24@ucl.ac.uk

Abstract

This report presents a reinforcement learning agent for a 4×4 four-in-a-row variant of Tic-Tac-Toe. A tabular Q-learning agent is trained through a curriculum of random-opponent play followed by self-play, using an ϵ-greedy exploration schedule with linear decay. Board symmetries (rotations and reflections) are exploited to reduce the state space and accelerate learning. A value-iteration agent is included as a deterministic baseline. After training for 500,000 episodes, the Q-learning agent achieves a win rate of approximately 0.98 against a random opponent and attains a near-perfect win rate against the value-iteration baseline in head-to-head evaluation. Ablation experiments show that the curriculum schedule and exploration decay rate are the most influential factors in final performance, while a hyperparameter sweep over learning rate and discount factor confirms stable convergence across a range of settings. Human-play benchmarks reveal occasional weaknesses in defensive play, motivating future work on improved opponent modeling and stronger tactical awareness.Human-play benchmarks reveal a strong tendency toward draw outcomes, likely due to the limited strategic complexity of the 4×4 setting, motivating further investigation into richer environments. To address this limitation, we additionally explore a larger-scale challenge setting of 9×9 five-in-a-row, demonstrating the potential for more strategic depth and improved learning dynamics in expanded state spaces.

Contents

1 Introduction

In this project, our main goal is to develop reinforcement learning agents to play a 4×4 variant of Tic-Tac-Toe where two players alternately place marks on an empty board until one achieves four in a row or the board is filled.

The primary focus of the project is the implementation of a tabular Q-learning agent that learns through interaction with the environment, improving its policy over time via experience. To provide a baseline for comparison, a Value Iteration agent is also implemented.

The agents are evaluated using several benchmarks, including head to head comparison between Q-learning and Value Iteration, qualitative assessment through interactive human play and win rate against opponents that move randomly. Additional ablation studies and hyperparameter sweeps are used to analyse how design choices such as opponent curriculum and exploration scheduling affect learning performance.

The game environment is made as a Python class which maintains a 4×4 board, tracks the current player and provides methods to reset, apply actions and query game status. States are encoded as flat or tuple representations indicating empty cells or marks (X/O) and actions correspond to placing marks in legal positions. The environment checks for wins and draws at each step and returns rewards accordingly, supporting both self-play and matches against fixed opponents.

2 Agent

In order to develop an agent capable of playing 4x4 Tic-Tac-Toe, two reinforcement learning methods, Q-learning and Value Iteration, were identified as particularly suitable. These methods were chosen because they provide a useful contrast between model free reinforcement learning and model based planning. Q-learning improves through repeated interaction with the environment and updates action values from observed transitions, whereas Value Iteration computes a value function offline by exploiting the known game dynamics.

The Q-learning agent is the primary reinforcement learning algorithm studied in this project. It is designed to estimate the quality of state-action pairs through experience and gradually improve its policy through temporal-difference updates. By contrast, the Value Iteration agent serves mainly as a deterministic planning baseline, allowing the learned Q-learning policy to be compared against a method that performs explicit dynamic programming over reachable states.

2.1 Q-Learning Agent

The agent is a tabular Q-learning agent playing as Player 1 (`X`) on the 4×4 board. Board states are encoded as $+1$ (X), -1 (O) and 0 (empty) and the action space at any state is the set of legal empty cells, identified by $(\text{row}, \text{column})$. The learned state representation also records which player is to move (so the same board with a different side to act is treated as a different state).

The Q-function is stored as a dictionary mapping $(\text{state}, \text{action})$ pairs to scalar values. Updates follow the standard TD rule, using the terminal reward $r \in \{+1, 0, -1\}$ at game end and a bootstrapped target $\gamma \max_{a'} Q(s', a')$ otherwise, always from Player 1's perspective. During evaluation the agent acts greedily, selecting $\arg\max_a Q(s, a)$ with ties broken at random.

To reduce the state space, boards related by rotation or reflection are mapped to a single canonical form before any Q-table lookup or update, with actions remapped back to the original board. This shrinks the Q-table and accelerates learning.

2.2 Value Iteration Agent

In this implementation, each state is defined by two components: the board configuration and the current player to move. This is important because the same board position may have a different value depending on whose turn it is. For example, a board that is favourable for Player 1 may be unfavourable if it is Player 2's turn, so the agent explicitly treats these as separate states. The board is represented in the same 4×4 format as the environment, with cells encoded as +1 for X, 1 for O, and 0 for empty. The valueiteration agent is a deterministic planning baseline rather than a learning agent in the usual reinforcementlearning sense. Instead of improving through sampled experience, it enumerates reachable states and repeatedly applies Bellman backups to estimate the optimal value of each state. For each state, the agent considers

all legal actions, evaluates the resulting successor states, and updates the current value estimate according to the best available action.

Because the game is finite and deterministic, the discount factor is set to γ=1.0. This is appropriate for a short episodic game such as TicTacToe, where the only meaningful outcomes are win, draw, or loss, and there is no need to prefer earlier rewards through discounting. The implementation also uses a convergence threshold of $\theta = 10^{-4}$, which stops the iterative updates once the maximum change in state value becomes sufficiently small. To keep computation manageable, the search is limited to 800000 reachable states and a maximum of 12 full sweeps over the state space. As with the Qlearning agent, the valueiteration baseline is trained from the perspective of Player 1, so the resulting policy is aligned with the same reward convention used elsewhere in the project.

3 Training

3.1 Q-Learning Agent

The opponent during training is configurable. By default a curriculum is used, in which the first 60% of episodes pit the agent against a uniformly random opponent to encourage broad state coverage, after which self-play begins. Player 2 is controlled by the same Q-table (with its own exploration) to expose the agent to a stronger and a more structured counterplay.

Exploration follows an ε-greedy schedule: ε decays linearly from 1.0 down to 0.01 over the first 400,000 episodes, which ensures that the agent explores widely early on before committing to a greedy policy as the Q-values stabilise. Because the game alternates turns, a perspective-correction step is applied after each Player 1 move so that the bootstrapped target always reflects the value from Player 1's point of view. This keeps the Q-learning updates internally consistent across alternating turns.

The core hyperparameters were as follows: a learning rate of $\alpha = 0.1$, a discount factor of $\gamma = 0.95$, ε decaying from 1.0 to 0.01 over 400,000 episodes, 500,000 total training episodes and a fixed random seed of 42 for reproducibility.

After the first episode and every 10000 episodes, the agent is evaluated over 200 games against a random opponent and the episode number, current ε, win/draw/loss counts, win rate, Q-table size and opponent type are written to a .csv log. The final Q-table is serialised as a file for use in the demo and submission.

3.2 Value Iteration Agent

The value iteration agent is trained as an offline planning baseline rather than by interacting with the environment through episodes. Instead of learning from sampled gameplay, it systematically enumerates reachable game states, updates their values using Bellman backups, and repeats this process until the value function stabilises or the iteration limit is reached.

In the implementation, the agent is configured with a discount factor of $\gamma = 1.0$, a convergence threshold of $\theta = 110^{-4}$, a maximum of 800,000 reachable states, and a maximum of 12 full iterations over the state space. A progress interval of 10 is also used so the training process can report its status periodically while the value function is being updated. The agent is trained for a fixed role, with agent_player = 1, meaning the learned policy is built from the perspective of Player 1.

During each iteration, the agent sweeps through the discovered state space and updates state values based on the best available action from that state. Because the agent is planning offline, there is no exploration schedule, replay buffer, or learning rate in the usual reinforcement-learning sense. Instead, the quality of the policy comes from repeated value backups across the enumerated state graph until the maximum change across states becomes small enough, or the iteration cap is reached. In practice, the max_states limit prevents the state expansion from growing without bound, and max_iterations keeps the computation manageable for the coursework setting.

Once training is complete, the learned value-iteration policy is saved and the sweep history is written to logs. The saved log stores, for each sweep, the sweep number, the number of known states, the Bellman update magnitude Δ, and the number of newly discovered states. This makes it possible to inspect convergence behaviour after training and compare the offline planner against the learned Q-learning agent using the same evaluation framework.

4 Methodology

Several forms of evaluation were used. First, both agents were tested against an opponent that moves randomly in order to measure overall playing strength under a simple and reproducible benchmark. Second, the trained Q-learning agent was compared directly against the Value Iteration baseline in head-to-head matches. Third, additional human-play testing was used as a qualitative check on how well performance against scripted opponents translated to more deliberate gameplay.

To better understand which design choices contributed most to performance, ablation experiments and hyperparameter sweeps were also conducted. These experiments varied factors such as the opponent curriculum, exploration schedule, learning rate, and discount factor. By comparing the resulting learning curves and benchmark scores, the contribution of each component could be examined systematically rather than relying only on a single final model.

Reproducibility was supported by running experiments through command-line configurations with fixed random seeds and by storing outputs in a consistent format. Training logs were written to CSV files, trained policies were saved as weight files, and generated plots were used to summarise performance after training. This allowed results to be inspected, compared, and reproduced without rerunning every experiment from scratch.

4.1 Q-learning Agent Improvement

The agent design evolved iteratively during development. The initial version used a pure reinforcement learning setup, where the Q-learning agent was trained solely through self-play against a random opponent. While this approach achieved reasonable performance under the random benchmark, human-play evaluation revealed clear weaknesses in tactical reasoning. In particular, the agent frequently failed to take immediate winning moves or to block the opponent's imminent wins, indicating that these critical patterns were not reliably learned through exploration alone.

To address these limitations, a hybrid policy was introduced. The final agent combines learned Q-values with a small set of rule-based tactical checks applied at decision time. Specifically, before consulting the learned policy, the agent first (i) executes an immediate winning move if available, (ii) blocks the opponent's immediate winning move, (iii) creates fork opportunities when possible, and (iv) attempts to prevent opponent forks. Only when none of these conditions apply does the agent fall back to the learned Q-policy. This design preserves the adaptability of reinforcement learning while ensuring basic tactical competence in critical situations.

Importantly, this hybridisation does not alter the underlying learning process but instead acts as a decision-time safeguard. As such, evaluation distinguishes between the raw learned policy and the hybrid policy to ensure that improvements due to reinforcement learning are not conflated with the contribution of heuristic rules.

5 Code Structure

The project is organised around a clear separation between environment dynamics, agent logic, training and evaluation. workflows, visualisation utilities, and the interactive user interface.

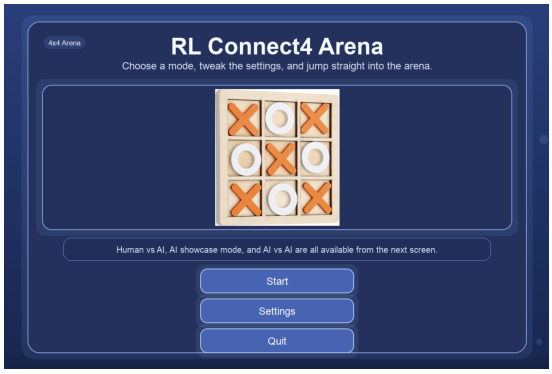
The environment layer acts as the single source of truth for the 4x4 Tic-Tac-Toe game dynamics. It defines board initialisation, legal move generation, state transitions, turn switching, terminal checks and winner reporting. By centralising these mechanics in one environment implementation, all agents and training scripts operate under the same rules, ensuring consistency across learning, evaluation, and interactive play.

The agent layer contains two main implementations. The tabular Q-learning agent that handles Q-table management, action selection, temporal-difference updates, symmetry-based canonicalisation of equivalent board states, and model serialisation for saving and loading learned policies. The value iteration agent provides a model-based planning baseline by performing offline dynamic programming over reachable game states. Its implementation handles state indexing, transition caching, Bellman backups, convergence tracking, and persistence of the computed value function. Together, these two agents provide both a learning-based and a planning-based approach within the same framework.

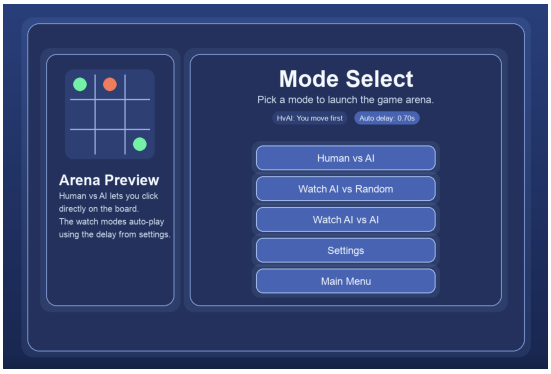
The training layer is separated from the agent definitions to enable reproducible experimentation. The Q-learning training script manages the episode loop, opponent selection, exploration scheduling, reward updates, periodic evaluation, and the export of learned weights and CSV logs. The value iteration training script configures planning hyperparameters, runs the sweep-based optimisation process, stores the resulting value table, and logs convergence statistics such as state counts and Bellman error. Dedicated evaluation utilities provide reusable functions for comparing agents against random opponents or against each other, while automation scripts support ablation studies and hyperparameter sweeps.

The visualisation layer is used to analyse training outcomes after experiments have completed. Plotting scripts read CSV logs produced during Q-learning or value iteration and generate learning curve figures, while a separate comparison plotting utility summarises agent match results.

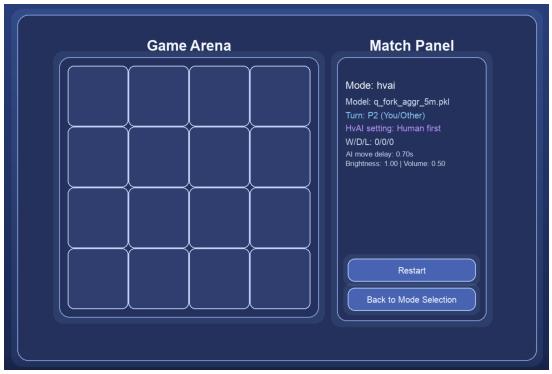
To support direct interaction with the trained Q-learning agent, a separate Pygame-based application for the 4x4 game was implemented. At startup, it loads the saved Q-learning policy and initializes the game environment. It then presents a sequence of interface scenes shown in Fig. 1 including the main menu, mode selection, settings, and gameplay screens. Within this interface, the program manages user input, button interactions, match configuration, board rendering, and game state updates, allowing either a human player or automated opponents to interact with the trained agent through a simple graphical front end.



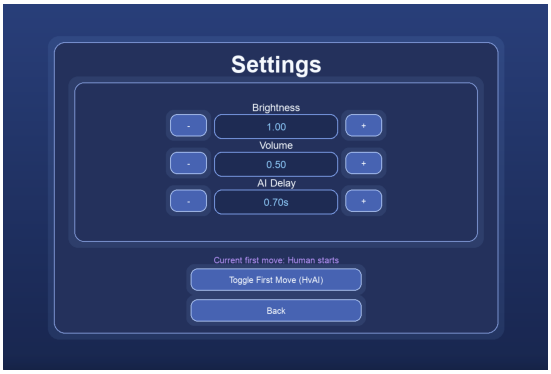
(a) Main Menu



(b) Mode Selection



(c) Gameplay Screen



(d) Settings

Figure 1: UX/UI of 4x4 Tic-Tac-Toe

6 Benchmark and Analysis

6.1 Ablation Results

Three ablations were run to understand which design choices matter most. Switching from a curriculum opponent to a random-only opponent throughout training resulted in lower final win rate (curriculum: 0.8000, random-only: 0.7400), suggesting the curriculum does meaningfully improve coverage. A faster ε decay converged more quickly but reached a higher final win rate (fast: 0.7600, slow: 0.8400), indicating that sufficient early exploration is not important. Compared with a faster schedule, a slower ε decay

produced clear gains in final performance, which implies a trade-off between sample efficiency and long-run policy quality.This behaviour is likely due to the relatively small state space, where more thorough exploration improves policy robustness.

6.2 Hyperparameter Analysis

A sweep over $\alpha \in \{0.1, 0.5\}$ and $\gamma \in \{0.95, 0.99\}$ showed that the best final win rate was achieved with alpha=0.1, gamma=0.95 (final win rate: 0.8000). Comparing learning-rate settings, the $\alpha = 0.5$ group achieved lower final performance than $\alpha = 0.1$ on average (avg: 0.6600 vs 0.8000). For discount factor, the effect was negligible, with $\gamma = 0.99$ vs $\gamma = 0.95$ averages of 0.7300 and 0.7300, respectively. A subsequent hyperparameter sweep was conducted to further refine the results. Based on this, the learning rate was fine-tuned to $\alpha = 0.12$ and the discount factor to $\gamma = 0.99$, which yielded the best overall performance. Compared to the initial sweep, this refined setting provided a slight improvement over $\alpha = 0.1$, confirming that a moderately low learning rate is beneficial for stable convergence. The effect of the discount factor remained relatively small, though $\gamma = 0.99$ was ultimately selected for its marginally better long-term performance.

6.3 Learning Curves

At each checkpoint the current policy plays 200 games against a random opponent, with win, draw and loss counts and overall win rate logged to CSV. Checkpoints occur after the first episode and then every `eval_every` episodes thereafter (10,000 in the default configuration). All runs use a fixed seed of 42 so results are directly comparable across experiments.

Figure 2 shows the learning curve of the original pure Q-learning agent, trained solely through self-play against the random opponent. The curve captures the growth of the agent's win rate as the Q-table is populated and the policy is refined.

Figure 3 shows the learning curve of the hybrid agent, which extends the original Q-learning policy with tactical heuristics such as immediate winning moves, opponent blocking, and fork handling. This hybrid approach preserves the learned Q-values while ensuring basic tactical competence, resulting in a stable and robust policy.

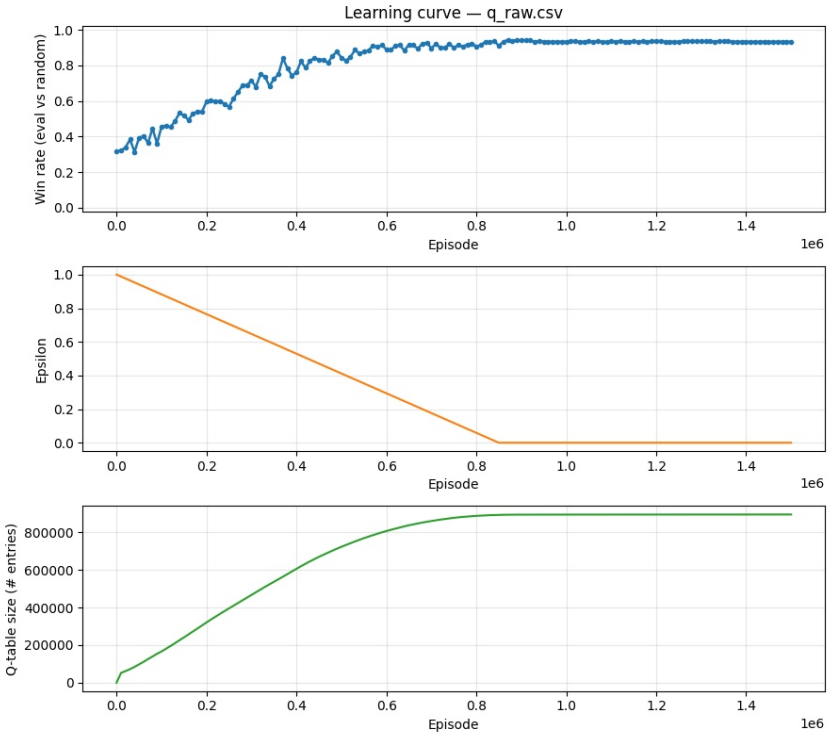


Figure 2: Win rate against a random opponent for the pure Q-learning agent over training episodes.

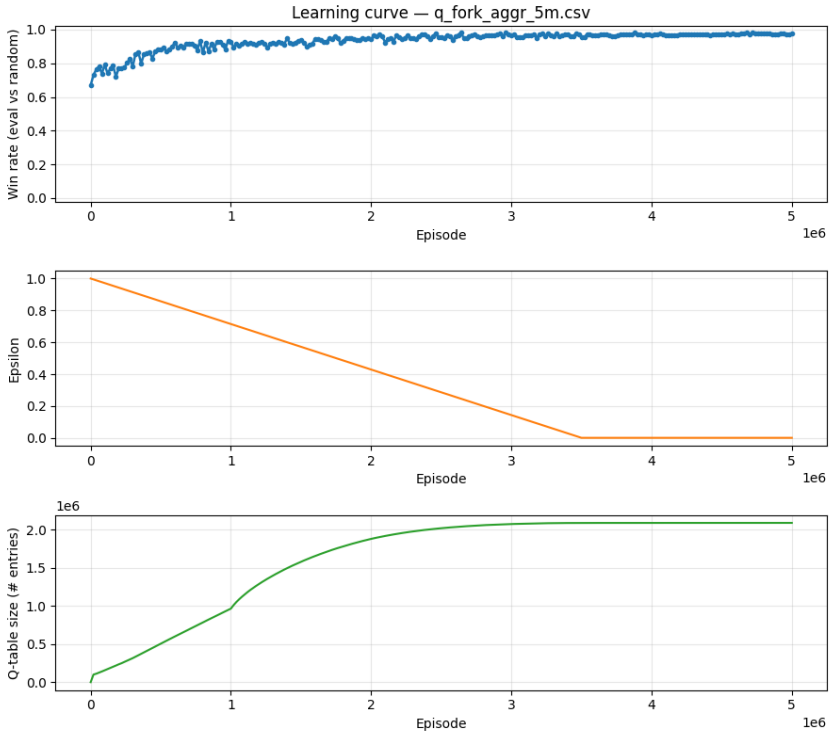


Figure 3: Win rate against a random opponent for the hybrid agent, combining learned Q-values with tactical heuristics.

The pure Q-learning agent (Figure 2) gradually improves its win rate against the random opponent, reaching a plateau close to 0.95 after approximately 1.2 million episodes. Its Q-table grows steadily until it saturates, and the exploration rate ϵ decays linearly to zero. In contrast, the hybrid agent (Figure 3) starts with a significantly higher initial win rate due to the incorporation of tactical heuristics. Although it requires more episodes to fully populate the larger Q-table, it quickly reaches near-optimal performance, with a win rate close to 1.0 after around 3.5 million episodes. The ϵ decay and Q-table growth follow similar patterns, but on a larger scale corresponding to the extended state-action space. Overall, while they both start learning from zero knowledge, the hybrid approach accelerates early performance and stabilizes the policy by combining learned Q-values with heuristic guidance, indicating we have a correct evolution.

Figure 4 shows the learning curve of the Value Iteration (VI) benchmark. Unlike the Q-learning agents, VI performs full sweeps over the state space, updating value estimates according to the Bellman residual. The top plot shows the win rate against a random opponent, which remains relatively stable in the early sweeps and rapidly increases after sweep 10, eventually converging near 1.0. The middle plot displays the Bellman residual, which spikes during major updates and decreases as the value function converges. The bottom plot tracks the number of known states explored, illustrating the progressive coverage of the state space.

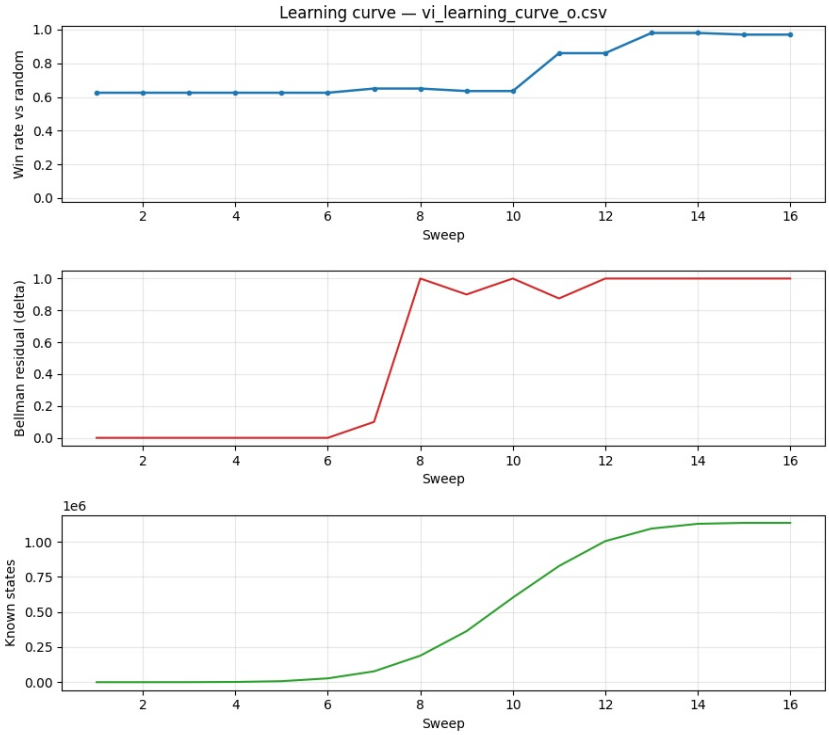


Figure 4: Win rate against a random opponent for the pure Q-learning agent over training episodes.

This learning curve is shown to prove that we have another value iteration agent which learns from zero knowledge and perform near perfect against random opponent after training. The value iteration agent provides a non-random benchmark opponent for comparison. It demonstrates how Q-learning agents perform against a policy that is not random, giving context to their learning progress and win rates.

6.4 Final Benchmark Results

To evaluate the performance of the trained Q-learning agent and Value Iteration agent, each agent was tested both against an opponent that moves randomly and against one another for 200 games with the results shown in Fig. 5 and summarised in Table 1

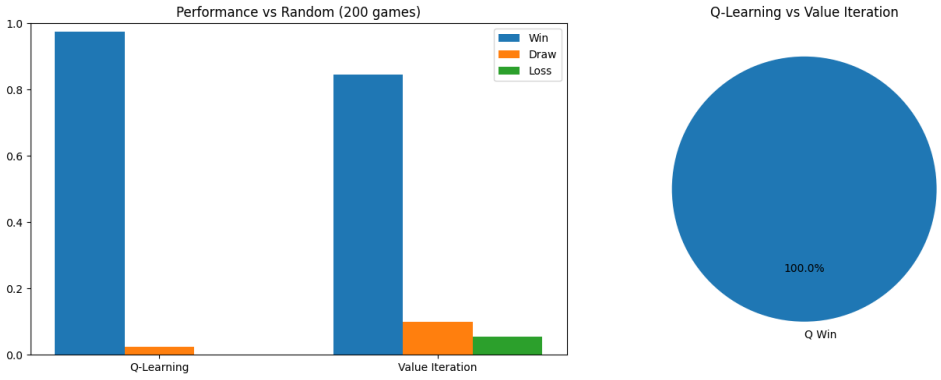


Figure 5: Win rate against a random opponent over training episodes.

As shown in Table 1, both the Q-learning agent and the Value Iteration agent achieve similar performance against opponents that move randomly, with win rates of 97.5% and 97% respectively. However, when evaluated against each other, the Q-learning agent clearly outperforms the Value Iteration agent, achieving a 100% win rate.

To further investigate the performance of each agent, additional evaluations were conducted against

Table 1: Final benchmark results over 200 games per matchup.

Agent	Opponent	Wins	Draws	Losses	Win rate (%)
Q-learning	Random	195	5	0	97.5
Q-learning	VI agent	200	0	0	100
VI baseline	Random	194	6	0	97

human players, allowing for a more practical assessment of gameplay strength beyond simulated environments, with each agent–opponent pair playing 20 games. The results are shown in Table 2

Table 2: Benchmark against Human results over 20 games per matchup.

Agent	Opponent	Wins	Draws	Losses	Win rate (%)
Q-learning	Thun	6	14	0	30
Q-learning	Sofiya	3	17	0	15
VI baseline	Thun	4	13	3	20
VI baseline	Sofiya	2	12	6	10

6.5 Failure Cases and Limitations

The agent has clear limitations. The most prominent one observed during human-play evaluation is a strong tendency toward draw outcomes. This is a consequence of the constrained 4×4 grid. A moderately competent human player can consistently steer games toward draws by occupying key positions. Informal human-play testing suggests the agent can be beaten with consistent pressure, particularly in defense. Beyond draws, the tabular Q-learning formulation is bounded by the state space it has explored during training. The agent does not generalise to unseen board configurations, it cannot adapt to novel opponent strategies that fall outside its learned experience. This also means the approach does not scale naturally to larger boards without an exponential growth in the Q-table. That is what motivates the DQN-based extension explored later on.

6.6 Key Takeaways

The results demonstrate that tabular Q-learning with a curriculum training schedule and symmetry reduction is sufficient to achieve near-optimal performance against random and value-iteration opponents on a 4×4 board. However, the 4×4 setting introduces a structural ceiling on competitive depth. Human-play benchmarks reveal that draw outcomes are prevalent, not because the agent plays poorly, but because the grid is too small to support strategic variety. This points to the need for richer environments, where stronger opponents and more complex state spaces can drive more meaningful learning dynamics. This was later explored in the Gomoku challenge taslk.

7 Conclusion

This project implemented a complete reinforcement-learning pipeline for a 4×4 four-in-a-row variant of Tic-Tac-Toe, including environment design, tabular Q-learning, baseline value-iteration comparison, training automation and log-based analysis tooling. The final agent demonstrates clear improvement from near-random play to a substantially stronger policy under the chosen training configuration.

The learning curves show a consistent upward trend in win rate against a random opponent, reaching 98% after 5000000 episodes. Benchmark results further indicate competitive performance against VI Baseline, with 200/0/0 W/D/L summary. And against human players, with 9/31/0 W/D/L summary.Ablation and hyperparameter studies suggest that curriculum opponent are the most influential factors in both convergence speed and final policy quality, while α and γ mainly affect stability and sample efficiency within the tested ranges. The learning rate (α) mainly affects convergence stability, while the discount factor (γ) has a minor impact on long-term performance within the tested range.

Despite these strengths, several limitations remain. High performance against random or fixed opponents does not necessarily transfer to robust human-level play and informal testing indicates exploitable defensive weaknesses in some board configurations. In addition, the tabular formulation is practical for 4×4 but does not scale naturally to larger boards without significant growth in state-action complexity.

Overall, the project meets the coursework objective of training an autonomous agent that learns from interaction and improves over time. Future work should focus on stronger evaluation protocols, broader human-play benchmarking and function approximation methods.

8 Challenge Task: Gomoku (9x9 board)

8.1 Agent

To tackle the extra challenge task, we created a 9×9 Gomoku agent using deep Q-networks. The game environment supports 9×9 boards with win conditions of five consecutive stones in any row, column, diagonal or anti-diagonal. Rewards are +1 for winning, 0 for a draw and 1 for a loss, with intermediate steps yielding 0. The agent encodes the board stats as a 3-channel tensor: one channel for the agent's stones (1), one for the opponent's (1) and one for the current turn (1.0 or 0.0), which is fed into a convolutional Q-network.

The Q-network consists of three convolutional layers. They have $3 \rightarrow 64 \rightarrow 128 \rightarrow 128$ filters with a kernel size of 3 and a padding of 1. They are then followed by flattening, two fully connected layers and ReLU activations. The output dimension of the layers match the 9×9 action space. Agent training uses a replay buffer size of 300000, an Adam optimiser with a learning rate of $2e^{-4}$, batch size of 128, γ of 0.99, and a smooth L1 loss. The target network updates every 1500 steps, with gradient clipping at norm 5.0. Action selection combines a masked greedy policy using a candidate mask to focus on local threats and reduce invalid moves.

A key design feature incorporated into the agent is a tactical move override. Before the epsilon-greedy Q-selection, the agent checks for immediate wins or opponent wins using line-counting heuristics. If a win exists, it plays there with a 20% exploration noise; otherwise, it blocks the opponent's wins. This ensures the DQN learns stable policies without being overpowered by simpler threats. Candidate actions are generated with a window of radius 2 around existing stones (or the centre if the board is empty), and legal actions are masked in Q-values with a larger penalty.

8.1.1 Training Curriculum

Agent training runs 200000 episodes by default in a three-phase curriculum against varying opponents. This helps the agent build broad coverage before strength. In phase 1 which is between the 0% and 20% mark of episodes, the agent faces 100% random opponents only. This allows it to explore the state space. In phase 2 which is between the 20% and 60% mark of episodes, it faces 45% of random opponents, and 55% of heuristic opponents. Finally, in phase 3 which is between the 60% and 100% mark of episodes, the agent faces 20% of random opponents, 35% of heuristic opponents, and 45% of playing against itself with a reduced ϵ of 0.05.

Epsilon decays from 1.0 to 0.05 over 120000 episodes, with the value updating every 4 steps. Evaluation occurs every 5000 episodes against the heuristic opponent. The first/second win rates, overall win rate, average loss, replay size and opponent type are all recorded in the process.

8.1.2 Heuristic Opponents

A rule-based heuristic bot serves both as a training opponent and an evaluation benchmark. It prioritises immediate wins, then blocks opponent wins, then scores actions by line patterns. Higher-likelihood winning lines gain more score, such as 4-in-a-row with both sides being open ends gains a significant amount of score, where as 3-in-a-row with one open end gains significantly less. Open ends and centre bias are factored in, with candidates limited to a window with radius 2 around each stone. This bot is strong enough to pressure the DQN agent during training but simple enough to beat reliably after convergence.

8.2 Code Structure

The 9×9 Gomoku extension keeps the same modular structure as the 4×4 project: environment, agent, training/evaluation, visualisation, and UI are separated.

The environment module is the single source of truth for Gomoku rules, including legal moves, turn switching, terminal checks, and five-in-a-row winner detection.

The agent module implements a DQN-based Gomoku policy with CNN state encoding, replay training, target-network updates, and masked legal action selection. It supports both a raw RL mode and a hybrid mode with tactical overrides (immediate win/block checks).

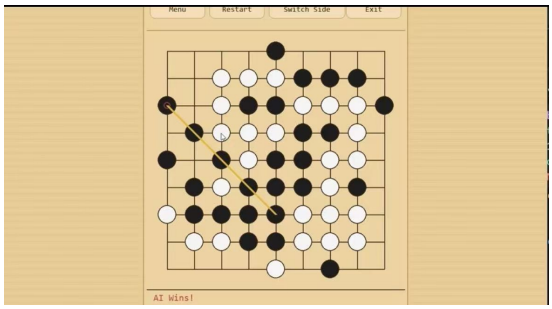
The training/evaluation scripts handle curriculum opponent scheduling (random/heuristic/self-play), epsilon decay, checkpoint saving, and CSV logging (e.g., `first_wr`, `second_wr`, `overall_wr`, `avg_loss`). Benchmark utilities run standardized matchups between raw, hybrid, and heuristic agents.

The visualisation utilities generate result plots such as overall win-rate comparisons, first-vs-second performance charts, and W/D/L distributions.

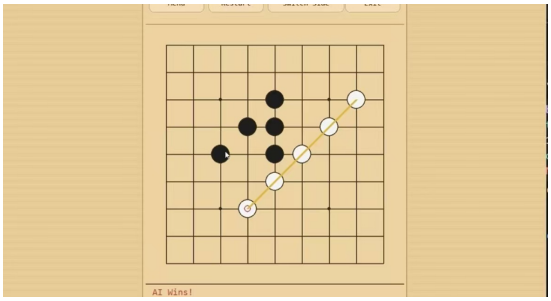
A Pygame UI is provided for interactive play with the trained Gomoku agent, including menu, settings, and gameplay screens.

(a) Main Menu

(b) Mode Selection



(c) Gameplay Screen



(d) Settings

Figure 6: UX/UI of 9x9 Gomoku

8.3 Benchmark and Analysis

The benchmark evaluates five matchups using the same evaluation protocol, reporting first-player win rate (first_wr), second-player win rate (second_wr), and overall win rate (overall_wr). The tested agents include: (i) a heuristic bot baseline, (ii) a raw RL policy (`agent_raw`), and (iii) a hybrid policy (`agent_hybrid`) that combines RL with tactical rule constraints.

Matchup	Overall Win Rate (WR)
heuristic_bot vs random	1.0000
agent_raw vs random	0.9667
agent_hybrid vs random	1.0000
agent_raw vs heuristic_bot	0.0833
agent_hybrid vs heuristic_bot	0.9667

Table 3: Overall win rates for different agents against random and heuristic opponents.

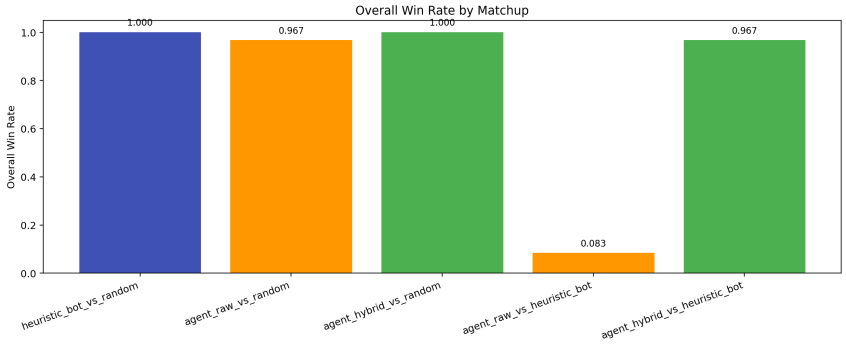


Figure 7: Overall win rates for different agents barchart

This indicates a clear pattern: while both learned agents perform well against random opponents, only the hybrid policy maintains strong performance against a tactical opponent.

The evaluation reveals a sharp robustness gap between the raw DQN policy and the hybrid policy under opponent shift. While the raw agent performs strongly against random play (overall win rate 0.9667), it collapses against the heuristic bot (0.0833), indicating poor tactical generalization beyond the training distribution. In contrast, the hybrid agent reaches 0.9667 against the same heuristic opponent. This suggests that end-to-end RL alone fails to reliably enforce critical tactical constraints (e.g., immediate block/forced defense), whereas adding rule-based guardrails removes catastrophic tactical errors and substantially improves out-of-distribution performance.

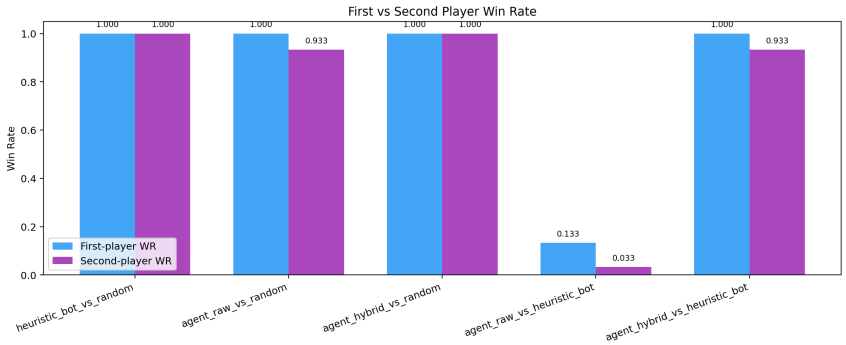


Figure 8: First vs Second Player Winning Rate Barchart

Figure 8 compares `first_wr` and `second_wr` across matchups and highlights both overall strength and side asymmetry. Against random opponents, both learned policies are strong as first player (1.0000), while second-player performance remains high (0.9333 for `agent_raw` and 1.0000 for `agent_hybrid`). The key failure appears in `agent_raw_vs_heuristic_bot`: first-player win rate drops to 0.1333, and second-player win rate further drops to 0.0333. By contrast, `agent_hybrid_vs_heuristic_bot` remains robust (1.0000 as first player, 0.9333 as second player).

The is likely happened since second player in Gomoku is structurally more defensive: it must react to initiative, absorb early pressure, and correctly answer forcing threats. This role amplifies tactical mistakes

because a single missed block can immediately convert into a forced loss. The raw RL policy has no hard tactical constraints (e.g., guaranteed must-block behavior), so it is more likely to choose moves that are globally plausible but locally losing. A heuristic opponent is specifically designed to exploit such tactical omissions, which explains why raw second-player performance collapses more severely than first-player performance. In contrast, the hybrid policy injects tactical guardrails (immediate win/block priorities and threat-aware filtering), making defense as second player substantially more stable.

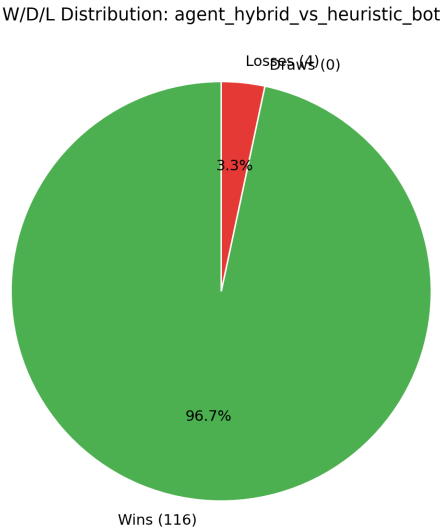


Figure 9: W/D/L Distribution: Hybrid DQN Agent vs Heuristic Bot

Figure 9 demonstrate the performance of trained hybrid DQN agent against heuristic bot,reporting 116 wins, 0 draws, and 4 losses, corresponding to a 96.7% win rate. This distribution indicates that the hybrid policy does not merely outperform the heuristic bot marginally; it dominates the matchup with consistently decisive outcomes.

First, the absence of draws (0) suggests that games are not drifting into neutral stalemates; instead, the policy frequently converts positions into concrete wins. Second, the very small loss count (4/120) implies that catastrophic tactical failures are rare. This is consistent with the hybrid design objective: preserve RL-driven strategic behavior while preventing high-impact tactical blunders. Third, compared with the raw policy collapse against the same opponent, this W/D/L profile provides strong evidence that the improvement is primarily due to robustness under adversarial tactical pressure, not random variance. Overall, Figure 9 supports the claim that hybridization substantially improves reliability and reduces exploitable error modes in out-of-distribution opponent settings.

9 Conclusion and Future Work

9.1 Conclusion

After completing the core 4×4 task, we still had remaining project time, so we challenged ourselves with a harder extension: 9×9 Gomoku. This extension is substantially more complex due to a much larger action space, longer tactical dependencies, and stronger sensitivity to opponent style. Despite these challenges, we successfully built and evaluated a complete DQN-based Gomoku system with curriculum training, tactical heuristics, and full benchmarking.

Our main contribution is not only achieving high win rate, but also demonstrating robustness under opponent shift. The raw RL policy performs well against random opponents (0.9667 overall WR) but collapses against the heuristic bot (0.0833). In contrast, the hybrid policy (RL + tactical guardrails) achieves 0.9667 against the same heuristic opponent, with 116/120 wins and only 4 losses. This confirms that in larger board games, end-to-end RL alone may learn strong in-distribution behaviour yet still fail on critical tactical constraints. Adding lightweight rule-based safety layers greatly improves reliability and practical play strength.

Overall, this challenge task demonstrates that our pipeline scales beyond tabular settings: from 4×4 tabular Q-learning to 9×9 deep RL with tactical integration, evaluation tooling, and GUI-based deployment. The system is functional, competitive, and technically well-validated.

9.2 Limitations

Although the 9×9 result is strong, it is not fully solved or theoretically optimal. Several limitations remain:

- **Dependence on tactical overrides:** strong performance currently relies on hybrid guardrails; pure raw policy still lacks robust tactical generalization.
- **Opponent coverage:** evaluation focuses on random and one heuristic family; broader adversaries may expose additional weaknesses.
- **Training instability:** RL learning curves are non-monotonic, and high performance checkpoints can fluctuate with opponent mixture.
- **Search depth:** the current policy is mostly one-step tactical; deeper lookahead is limited.

9.3 Future Work

There is clear room for improvement in both algorithmic strength and scientific rigor. We propose the following future directions:

1. **Stronger architectures:** test residual CNN backbones or lightweight transformer-style board encoders for better long-range pattern modeling.
2. **Better training objectives:** combine DQN with auxiliary threat-prediction heads or policy-improvement targets to improve tactical awareness.
3. **Self-play league training:** replace fixed opponent mixtures with league-style self-play and opponent sampling to reduce overfitting to one heuristic.
4. **Deeper tactical module:** add selective 2–3 ply threat-space search on top of Q-values, especially for forced win/defense lines.
5. **Broader benchmark protocol:** evaluate against multiple heuristic variants, unseen scripted bots, and different seeds; report confidence intervals.
6. **Larger boards:** extend experiments to larger boards such as 15×15 to study scalability and strategy adaptation.

In summary, the 9×9 challenge shows that our approach is already strong and practical, but still improvable. The current system provides a solid foundation for moving toward more general, stable, and tactically complete board-game RL agents.

10 Team contributions

Algorithm lead – Thun Sahacharoen
Environment engineer – Ted Tang
Data Analyst – Sofiya Flenova
Project manager/editor – Andrew Lau