National School of Computer Science and Systems Analysis -Rabat- (ENSIAS)

Artificial Intelligence Engineering (2IA)

Reinforcement Learning

Tic Tac Toe with Multiple Reinforcement Learning Agents

Realized by :
Ismail Eladraoui
Saâd Qacif

Professor :
M. Mohamed Naoum

November 6, 2025

Contents

List of Figures

1 Introduction

Tic Tac Toe is a classical two-player, zero-sum game that serves as an excellent testbed for exploring reinforcement learning (RL) algorithms. Despite its simplicity, the game provides a well-defined environment with a finite state space of approximately $3^9 = 19,683$ possible board configurations (or 5,478 accounting for symmetries). This makes it computationally tractable while still offering sufficient complexity to compare different learning paradigms.

The primary objective of this project is to design, implement, and systematically compare various agent strategies, ranging from non-learning heuristic approaches to sophisticated temporal-difference learning methods. Through this comparative analysis, we aim to understand the trade-offs between computational efficiency, learning capability, and optimality in game-playing agents.

1.1 Implemented Agents

We categorize our agents into three main groups:

Non-Learning Agents:

- Random Agent
- Exhaustive Search Agent
- Minimax with Alpha-Beta Pruning

Monte Carlo Methods:

- Monte Carlo Control

Model-Based Methods:

- Dynamic Programming (Policy Iteration, Value Iteration)

Temporal Difference Learning:

- SARSA (On-Policy)
- Expected SARSA
- Q-Learning (Off-Policy)
- Double Q-Learning

2 Theoretical Framework

2.1 Game as a Markov Decision Process

The Tic Tac Toe environment is formalized as a Markov Decision Process (MDP):

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$$

where:

- $\mathcal{S}$: Set of all possible board states (configurations of X's, O's, and empty cells)
- $\mathcal{A}(s)$: Set of legal actions (empty positions) available in state s
- $P(s'|s, a)$: State transition probability function. In Tic Tac Toe, this is deterministic given opponent policy
- $R(s, a, s')$: Reward function mapping state-action-next state triples to real values
- $\gamma \in [0, 1]$: Discount factor for future rewards

2.2 State Space Analysis

The complete state space $\mathcal{S}$ can be partitioned into valid and invalid board configurations.

Valid Board States: A board state s is valid if it satisfies the following constraints:

- $|n_X - n_O| \leq 1$, where n_X and n_O are the number of X's and O's respectively
- At most one player has achieved a winning configuration
- If X has won, then $n_X = n_O$ or $n_X = n_O + 1$
- If O has won, then $n_X = n_O$

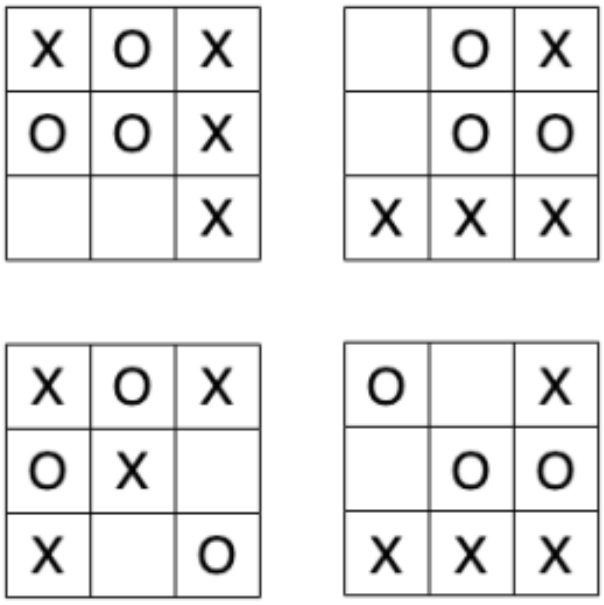


Figure 1: Examples of valid terminal-states(won by X)

Invalid Board States: States that violate any of the above constraints are unreachable through legal gameplay. Examples include:

- Both players having winning configurations simultaneously
- Difference in piece counts exceeding 1 (e.g., $n_X - n_O = 3$)
- O having more pieces than X (assuming X plays first)

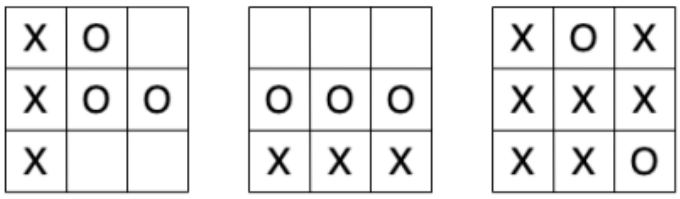


Figure 2: Example of invalid Tic Tac Toe board states. These configurations violate the game's rules and cannot be reached through legal play.

The total state space cardinality is $3^9 = 19,683$, but only approximately 5,478 states are valid when accounting for symmetries (rotations and reflections). After removing terminal states and symmetries, the effective state space for learning is further reduced.

2.3 Game Tree Representation

Tic Tac Toe can be represented as a game tree where:

- **Root node:** Empty board (initial state)
- **Internal nodes:** Non-terminal game states
- **Leaf nodes:** Terminal states (win, loss, or draw)
- **Edges:** Legal actions/moves
- **Depth:** Number of moves played (maximum depth = 9)
- **Branching factor:** Decreases as game progresses (9 initially, then 8, 7, ...)

Game Tree Properties:

- Maximum depth: 9 moves
- Average branching factor: ≈ 5
- Number of leaf nodes: 255,168 (including symmetries)
- Minimax complexity: $O(b^d) = O(5^9) \approx 2$ million nodes
- With alpha-beta pruning: $O(b^{d/2}) \approx 3,125$ nodes

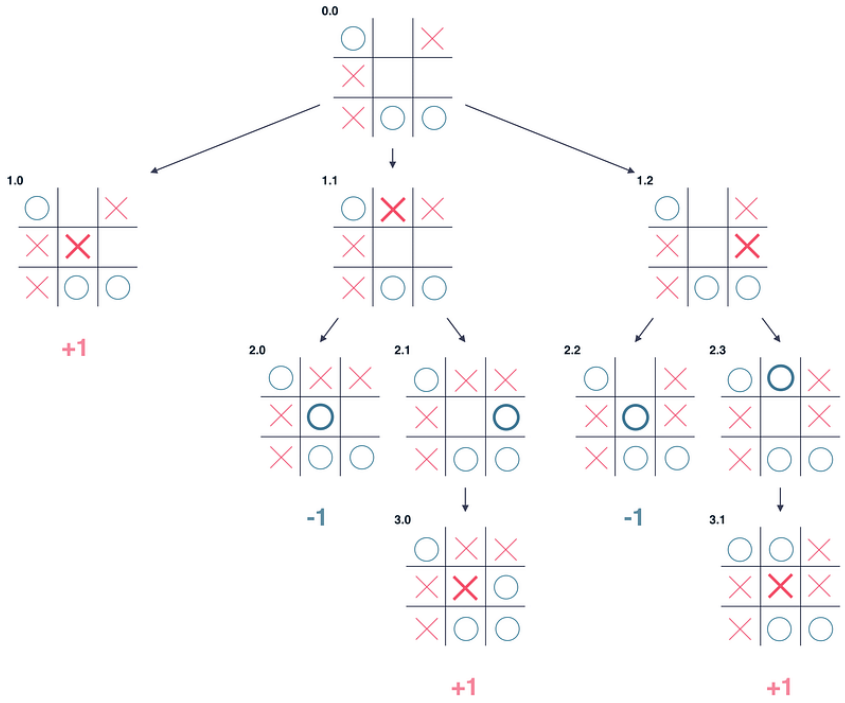


Figure 3: Partial game tree for Tic Tac Toe showing first two levels. The root represents the empty board, and each branch represents a possible move. The tree continues until reaching terminal states using +1, 0 and -1 as terminal utility values

2.4 Reward Structure

The reward function is sparse and terminal-based:

$$R(s, a, s') = \begin{cases} +1 & \text{if transition leads to agent victory} \\ 0 & \text{if game ends in draw} \\ -1 & \text{if transition leads to agent defeat} \\ 0 & \text{otherwise (non-terminal states)} \end{cases}$$

This sparse reward structure poses a challenge for learning algorithms, as credit assignment becomes difficult when rewards are delayed until episode termination.

2.5 Policy and Value Functions

A policy $\pi : \mathcal{S} \to \mathcal{A}$ maps states to actions. The state-value function under policy π is:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s\right]$$

The action-value function (Q-function) is:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a\right]$$

The optimal policy π^* maximizes the expected cumulative reward:

$$\pi^*(s) = \arg\max_{a} Q^*(s, a)$$

3 Non-Learning Agents

3.1 Random Agent

The random agent serves as a baseline, selecting actions uniformly from available moves:

$$\pi_{\text{random}}(a|s) = \frac{1}{|\mathcal{A}(s)|}, \quad \forall a \in \mathcal{A}(s)$$

Characteristics:

- No learning or state evaluation
- Constant time action selection: $O(1)$
- Win rate against itself: approximately 50% (50% draws)
- Useful for establishing performance baseline

3.2 Exhaustive Search Agent

This agent explores all possible game continuations from the current state to terminal nodes, counting win/loss/draw outcomes.

Algorithm: For each legal action a, compute:

$$\text{Score}(s, a) = \frac{\#\text{Wins} - \#\text{Losses}}{\#\text{Total outcomes}}$$

Then select: $a^* = \arg\max_a \text{Score}(s, a)$

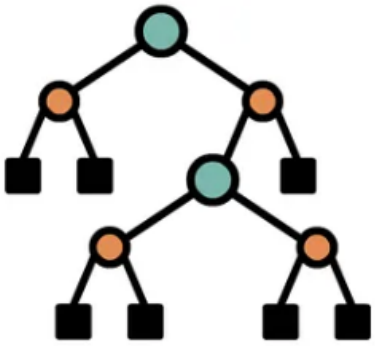


Figure 4: Exhaustive search tree exploration from a given state. The agent recursively explores all possible game continuations.

Complexity:

- Time: $O(b^d)$ where b is branching factor, d is depth
- Space: $O(d)$ for recursive calls
- Impractical for large games but illustrative for Tic Tac Toe

3.3 Minimax with Alpha-Beta Pruning

Minimax implements optimal adversarial search by assuming both players play perfectly.

Minimax Value:

$$V(s) = \begin{cases} \text{utility}(s) & \text{if } s \text{ is terminal} \\ \max_{a \in \mathcal{A}(s)} V(\text{result}(s,a)) & \text{if max player's turn} \\ \min_{a \in \mathcal{A}(s)} V(\text{result}(s,a)) & \text{if min player's turn} \end{cases}$$

Alpha-Beta Pruning: Maintains bounds α (best max can do) and β (best min can do). Prunes when:

$$\beta \leq \alpha$$

This optimization reduces the search space from $O(b^d)$ to $O(b^{d/2})$ in best case.

Properties:

- Guarantees optimal play in two-player zero-sum games
- Complete and optimal
- No learning required

4 Monte Carlo Methods

4.1 Monte Carlo Control

Monte Carlo methods learn from complete episodes without requiring a model of the environment.

First-Visit MC Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[G_t - Q(s,a)]$$

where the return G_t is:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} R_{k+1}$$

Exploration Strategy: We use ϵ-greedy policy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a'} Q(s,a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases}$$

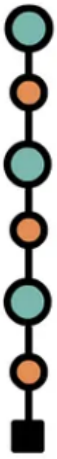


Figure 5: Monte Carlo episode trajectory showing complete state-action sequence from initial state to terminal outcome.

Advantages:

- Model-free learning
- Can learn from actual or simulated experience
- Unbiased estimates of value functions

Disadvantages:

- High variance in updates
- Requires complete episodes (slow for long games)
- Inefficient sample usage

5 Dynamic Programming

5.1 Model-Based Planning

Dynamic Programming assumes full knowledge of the MDP: transition probabilities $P(s'|s,a)$ and rewards $R(s,a)$.

Bellman Expectation Equation:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

Bellman Optimality Equation:

$$V^{*}(s) = \max_{a} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{*}(s')]$$

$$Q^{*}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a')]$$

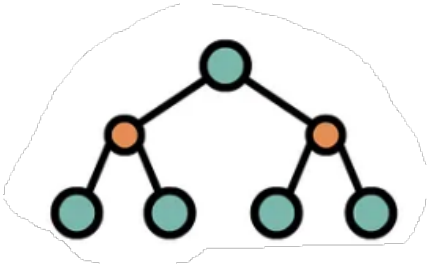


Figure 6: Dynamic Programming full-width backup diagram. From the current state (top green node), all possible actions lead to successor states (orange nodes), which further branch to future states (bottom green nodes). DP considers all transitions with their probabilities to compute exact value functions.

5.2 Policy Iteration

Alternates between:

1. **Policy Evaluation:** Compute V^{π} using iterative updates
2. **Policy Improvement:** Update policy greedily: $\pi'(s) = \arg\max_a Q^{\pi}(s,a)$

5.3 Value Iteration

Combines policy evaluation and improvement:

$$V_{k+1}(s) = \max_{a} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')]$$

Convergence: Guaranteed to converge to V^{*} as $k \to \infty$ under the contraction mapping property.

6 Temporal Difference Learning

TD methods combine advantages of MC (model-free) and DP (bootstrapping).

Figure 7: Temporal Difference one-step backup diagram. TD methods update the current state (top green node) using the immediate reward and the bootstrapped value of the next state (orange node), which itself estimates future rewards (bottom green node). This combines sampling (like MC) with bootstrapping (like DP).

6.1 SARSA (On-Policy TD Control)

SARSA updates Q-values using the actual next action taken:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Algorithm Flow:

1. Initialize $Q(s, a)$ arbitrarily
2. Choose a_t from s_t using policy derived from Q (e.g., ϵ-greedy)
3. Take action a_t, observe r_{t+1}, s_{t+1}
4. Choose a_{t+1} from s_{t+1} using same policy
5. Update $Q(s_t, a_t)$ using above equation
6. $s_t \leftarrow s_{t+1}$, $a_t \leftarrow a_{t+1}$

Properties:

- On-policy: learns value of policy being followed
- More conservative than Q-Learning
- Convergence guaranteed under standard stochastic approximation conditions

6.2 Expected SARSA

Instead of sampling next action, uses expected value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \sum_{a'} \pi(a'|s_{t+1}) Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Advantages over SARSA:

- Lower variance due to expectation vs. sampling
- Can be used on-policy or off-policy
- More stable learning in practice

6.3 Q-Learning (Off-Policy TD Control)

Q-Learning uses max operator, learning optimal policy while following exploratory policy:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

Key Properties:

- Off-policy: behavior policy can differ from target policy
- Converges to optimal Q^* under appropriate conditions
- More aggressive exploration possible
- Can suffer from maximization bias

6.4 Double Q-Learning

Addresses overestimation bias in Q-Learning by maintaining two independent Q-functions:

Update Rule: With probability 0.5:

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha[r_{t+1} + \gamma Q_2(s_{t+1}, \arg\max_{a'} Q_1(s_{t+1}, a')) - Q_1(s_t, a_t)]$$

Otherwise:

$$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha[r_{t+1} + \gamma Q_1(s_{t+1}, \arg\max_{a'} Q_2(s_{t+1}, a')) - Q_2(s_t, a_t)]$$

Intuition: By decoupling action selection (using one Q) from evaluation (using the other Q), we reduce positive bias from maximization.

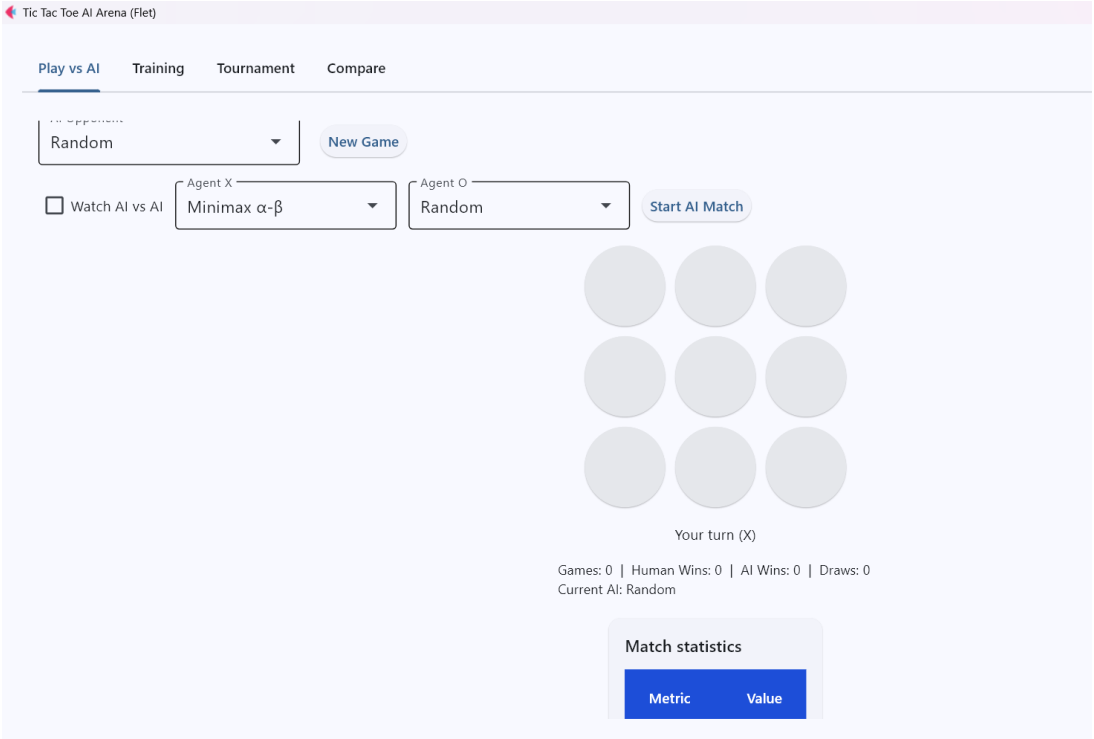
7 Implementation Details

7.1 Software Architecture

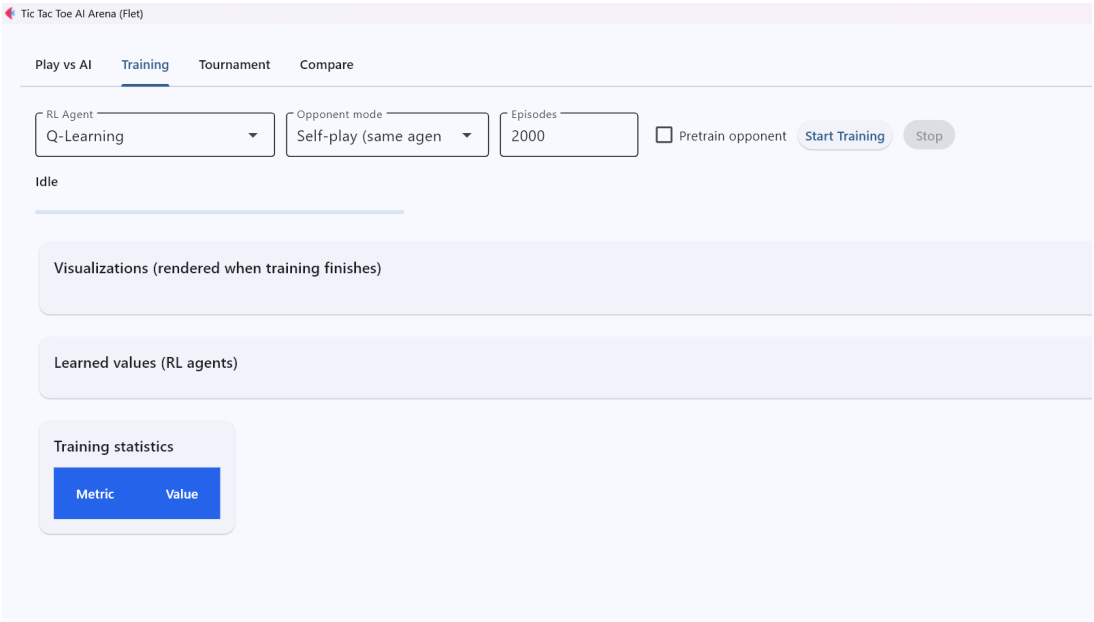
- **Programming Language:** Python 3.10+
- **Dependencies:** NumPy (numerical operations), Matplotlib (visualization), pickle (model persistence)
- **Environment Class:**
  - State representation: 3×3 NumPy array
  - Action space: integer indices 0-8
  - Reward calculation
  - Win/draw detection
  - State hashing for Q-table indexing
- **Agent Base Class:**
  - Abstract interface for action selection
  - Learning update method (for RL agents)
  - Policy extraction

7.2 Software Description

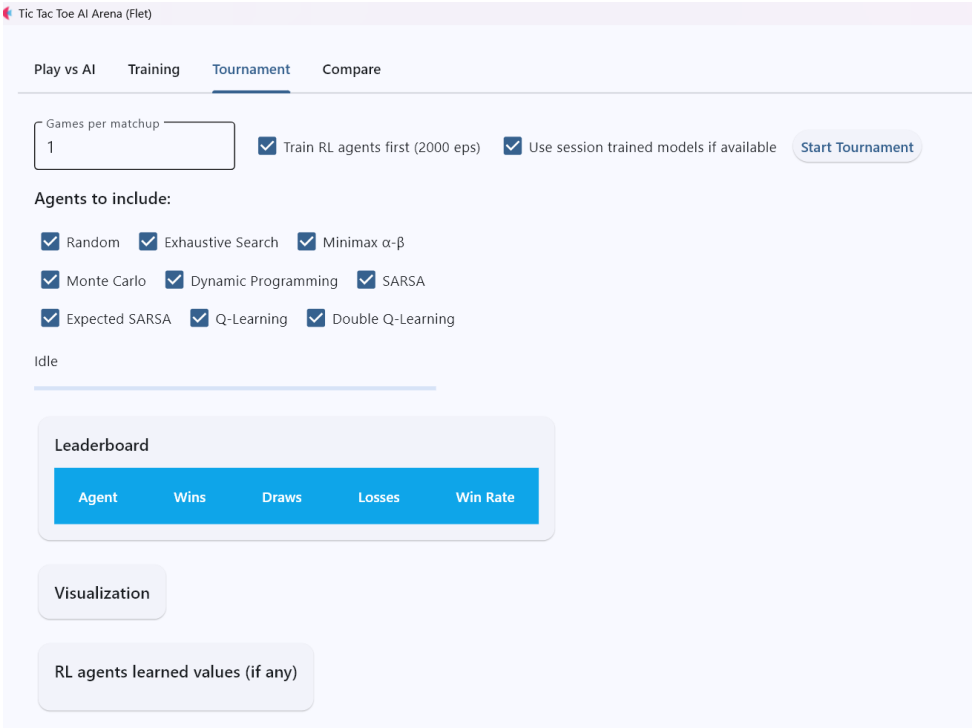
Tic-Tac-Toe is a desktop app built with Python and Flet that lets users play, train, compare, and tournament-test nine different Tic-Tac-Toe agents. The roster includes Random, Exhaustive Search, Minimax with alpha–beta pruning, Monte Carlo (every-visit MC control), Dynamic Programming (value-iteration planner), and the TD family: Q-Learning, SARSA, Expected SARSA, and Double Q-Learning. Everything runs locally with responsive UI, colorful tables, and compact visualizations.
The Play vs AI tab provides a live 3×3 board for Human vs AI and a "Watch AI vs AI" mode. While playing or spectating, the app updates a scrollable match statistics table that records sides, winner, moves, and agent names. The board scales with the window, and the session summary (games, human wins, AI wins, draws) stays visible as matches progress.



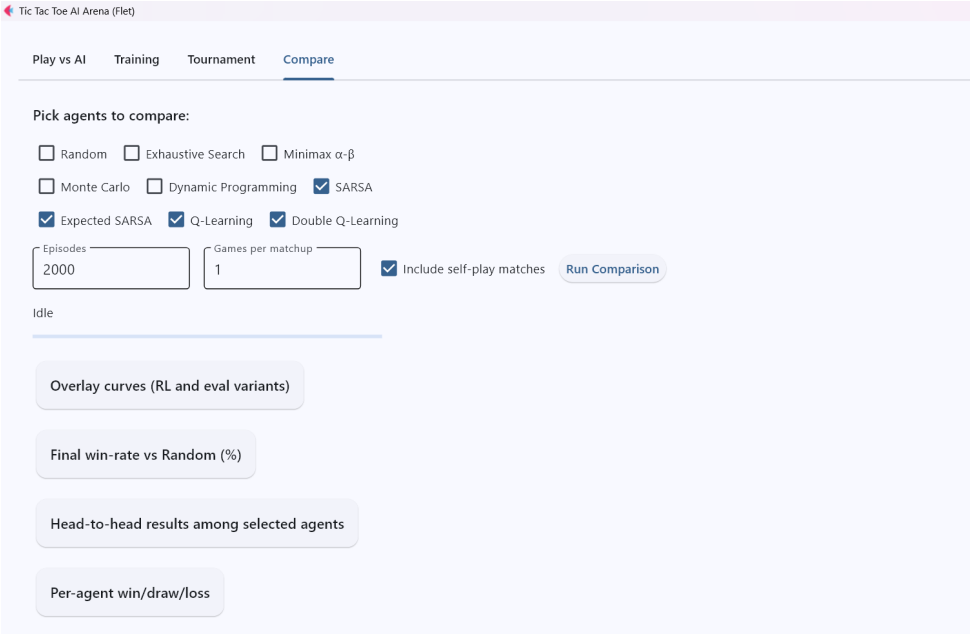
The Training tab turns the app into an RL lab. Any trainable agent—Q-Learning, SARSA, Expected SARSA, Double Q-Learning, Monte Carlo, or Dynamic Programming—can be trained either in self-play or against a chosen opponent, with an optional opponent pretraining step. Training can be stopped safely at any time. After a run, the app renders episode-wise Reward, Steps, Epsilon, and Q-table size plots, and shows learned-value visuals (Q-heatmap and Q-value histogram) for all RL agents including MC and DP. Stats are summarized in a styled table. Trained models are kept in memory for the current session and can be reused in other tabs.



The Tournament tab runs a round-robin among selected agents with configurable games per matchup, optional self-matches, and options to pretrain RL agents or reuse session-trained models. Results appear as a leaderboard with win rates and colored badges, plus an enlarged composite graphic: win-rate bars, win/draw/loss distribution, a head-to-head win-rate heatmap, and a games-distribution pie. Learned-value visuals for participating RL agents (including MC and DP) are collected below for quick inspection.



The Compare tab focuses on analysis. Users pick any subset of the nine agents, set an episode budget, and the app trains the selected RL agents to collect histories while evaluating non-RL agents for baselines. It then runs a round-robin among the selection (optionally including self-play) and displays three overlays (Reward, Steps, Epsilon), a head-to-head heatmap, per-agent win/draw/loss bars, a final "win rate vs Random" summary, and a compact per-agent table reporting average reward over the last 100 episodes, best MA-50, final epsilon, and Q-table size. A toggle reveals learned-value visuals for all selected RL agents, including MC and DP.



Under the hood, performance is tuned for large comparisons and tournaments. Minimax and Exhaustive Search use transposition tables to avoid re-evaluating positions, the DP planner caches transitions and rebuilds its Q-table only when a metric snapshot is recorded, and the tournament loop avoids allocations and logging in hot paths. These optimizations reduce typical 9-agent comparisons from minutes to seconds on a laptop CPU without changing behavior.

7.3 Hyperparameters

Parameter	**Value**
Learning rate (α)	0.1
Discount factor (γ)	0.95
Initial exploration (ϵ_0)	1.0
Final exploration ($\epsilon_{\min}$)	0.01
Exploration decay	0.9995
Training episodes	50,000
Evaluation episodes	1,000

Table 1: Hyperparameter settings for learning agents.

7.4 Training Procedure

1. Initialize Q-tables with zeros (optimistic initialization alternative: small random values)
2. For each episode:
   - Reset environment
   - Alternate turns between learning agent and opponent
   - Update Q-values after each transition (TD methods) or episode (MC)
   - Decay ϵ for exploration
3. Periodically evaluate against fixed opponents
4. Save learned Q-tables for later analysis

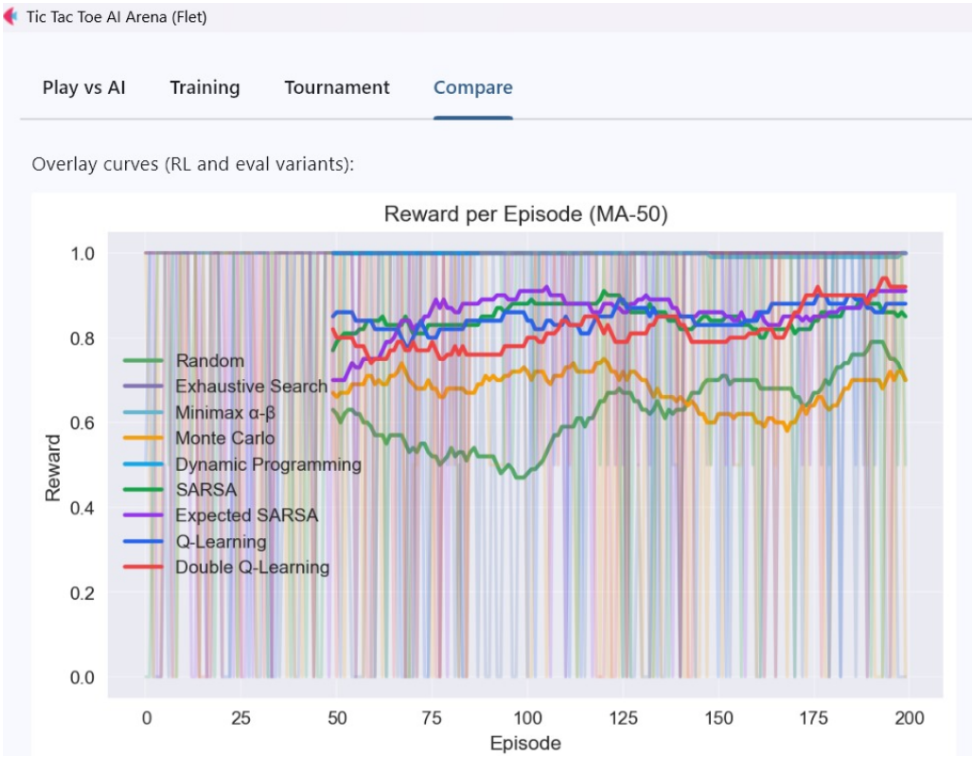
7.5 Evaluation Metrics

- **Win Rate:** Percentage of games won against specific opponents
- **Average Reward:** Mean cumulative reward per episode
- **Draw Rate:** Frequency of tied games
- **Loss Rate:** Percentage of games lost
- **Convergence Time:** Episodes until performance stabilizes
- **Q-Value Variance:** Measure of learning stability

8 Experimental Results

8.1 Performance Comparison

The bar chart compares the overall performance of each agent in terms of wins (green), draws (yellow), and losses (red) across all tournament matchups. The Minimax α-β and Exhaustive Search agents stand out with the highest number of victories and very few or no defeats, confirming the superiority of deterministic search-based methods in a fully observable game like Tic-Tac-Toe. Among the learning agents, Dynamic Programming achieves a balanced performance, while SARSA, Q-Learning, and Expected SARSA record a mix of wins and losses, showing that they partially approximate the optimal policy but still struggle against perfect opponents. Monte Carlo performs the weakest, suffering the most losses, which reflects its slower convergence and dependence on complete-episode returns. The Random agent sits near the bottom as a control baseline with low win counts, as expected. Overall, this chart visually ranks the agents from perfect planners to exploratory learners, revealing clear performance tiers within the mixed-paradigm tournament. This figure illustrates the evolution of the average reward per episode (MA-50) across all agents during the comparative evaluation phase. Each curve represents a moving average of rewards computed over a sliding window of 50 episodes, highlighting the rate and stability of convergence for each algorithm.



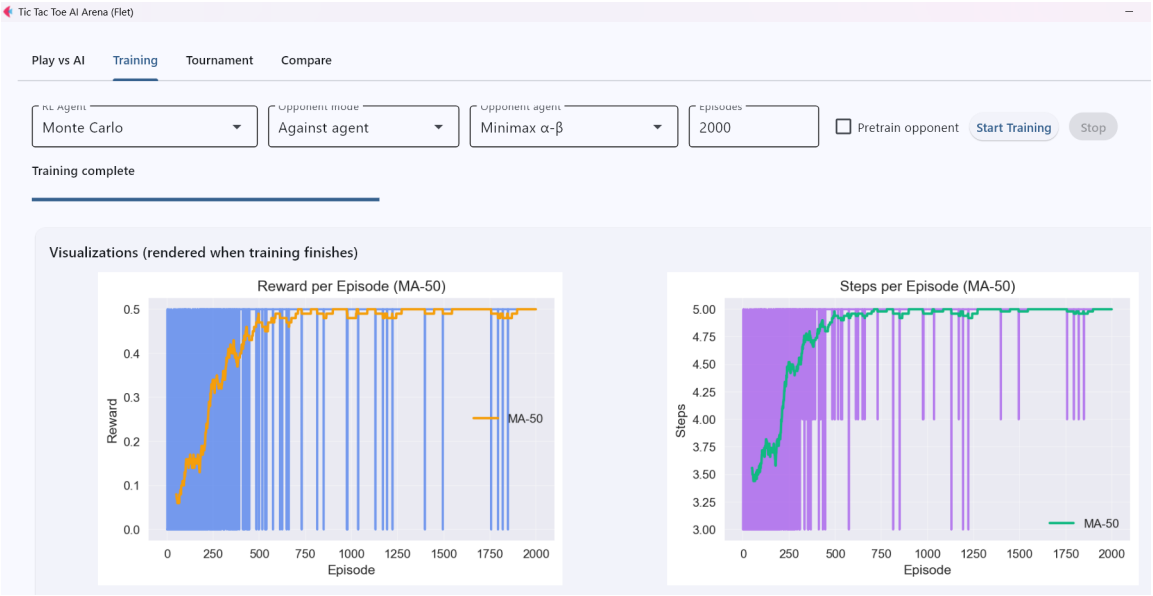
The **Minimax** α–β and **Exhaustive Search** baselines maintain consistently high reward values close to 1.0 throughout all episodes. This behavior is expected since both are deterministic, search-based agents that always compute optimal moves. Among the reinforcement learning algorithms, **Expected SARSA** and **Double Q-Learning** exhibit the most stable and highest convergence, reaching near-optimal reward levels after roughly 100 episodes. Their stability can be attributed to reduced overestimation bias and smoother value propagation.

The **Q-Learning** and **SARSA** agents also demonstrate steady improvement but with slightly higher variance, reflecting the influence of their exploration strategies and learning rate schedules. In contrast, **Monte Carlo** and **Dynamic Programming** settle on lower average rewards (approximately 0.6–0.8), which aligns with their limitations: DP assumes a random opponent model, and MC depends solely on complete-episode returns, making adaptation slower in deterministic environments. The **Random** agent remains near the bottom with an average reward around 0.5, serving as the baseline for uninformed play.

This line plot shows how the exploration rate ϵ evolves over time for all agents during training. For non-learning or fully deterministic methods such as Random, Exhaustive Search, Min max α-β, and Dynamic Programming, ϵ remains fixed at zero—these agents do not explore probabilistically. In contrast, the Double Q-Learning curve (in red) clearly demonstrates an exponential decay of ϵ, starting from full exploration ($\epsilon = 1.0$) and gradually reducing toward near-zero as the agent learns to exploit its acquired knowledge. This trajectory reflects the classic reinforcement learning trade-off between exploration and exploitation: early episodes emphasize discovery, while later episodes consolidate optimal decisions. The figure highlights that only the adaptive, experience-based agents adjust ϵ dynamically, whereas the search and model-based algorithms rely entirely on deterministic computation.

8.2 Training

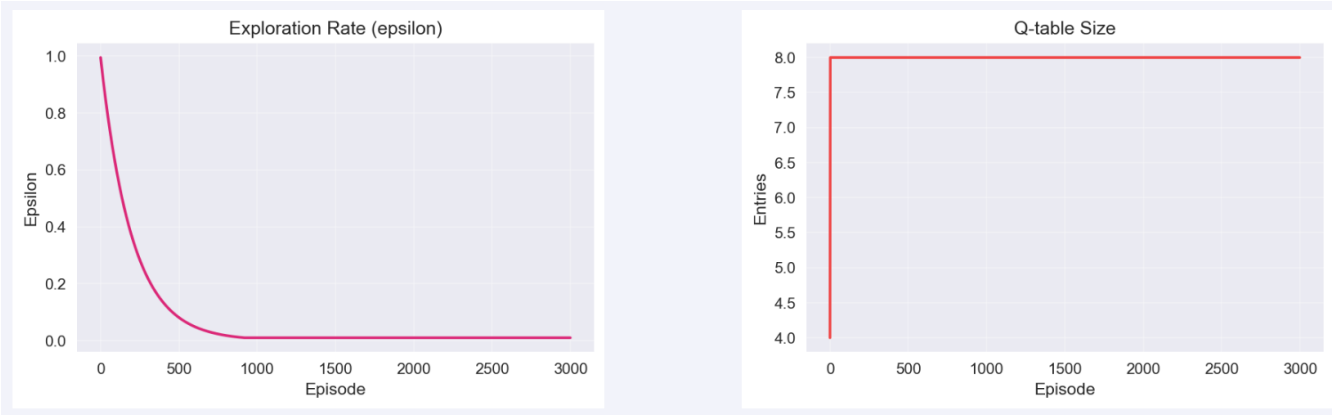
For the training part we used 2 options: training our agent against itself or against another agent. The trainable agents are Monte Carlo, Dynamic Programming, SARSA and its expected variant, as well as Q-Learning and Double Q-Learning. The examples shown here are Monte Carlo and Double Q-Learning.

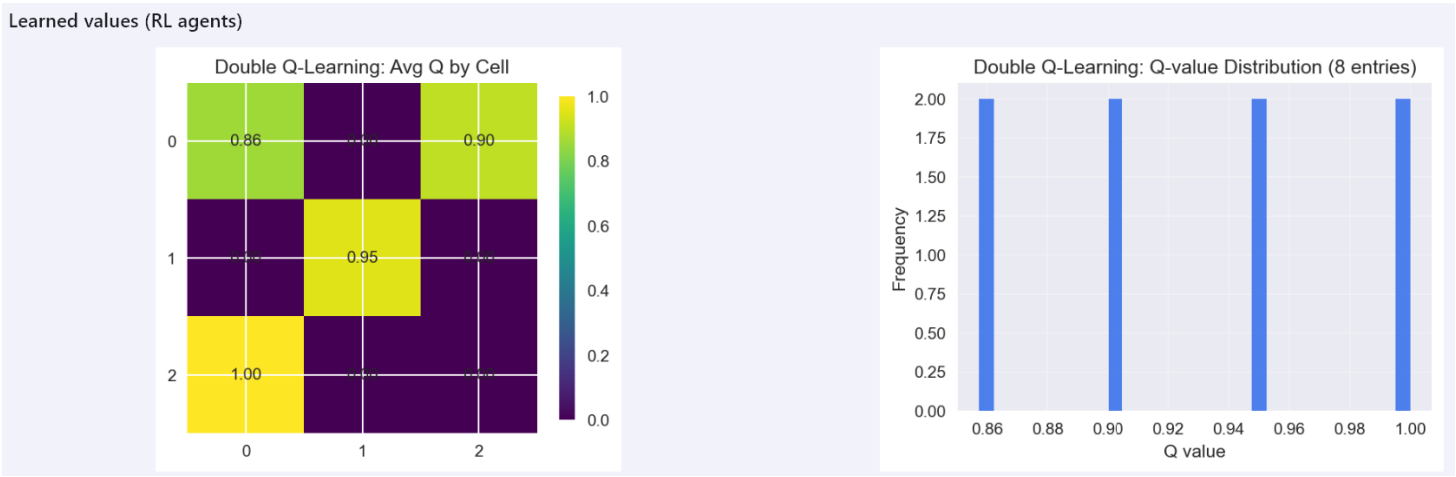


Training statistics

Metric	Value
Agent	Monte Carlo
Episodes	2000
Duration (s)	133.0
Final epsilon	0.010
Q-table size	257
Avg reward (last 100)	0.500
Avg steps (last 100)	5.00
Best MA-50	0.500

These results show the training behavior of the Monte Carlo reinforcement learning agent when playing Tic-Tac-Toe against a fixed Min max α-β opponent over 2,000 episodes.
Quantitatively, the training statistics confirm this convergence. The agent ran for 2,000 episodes over 133 seconds, ending with a very small $\epsilon = 0.010$, meaning exploration nearly stopped and the behavior became mostly greedy. Its learned Q-table contains 257 entries, covering a broad portion of the reachable state–action space. Both the average reward and best moving average (MA-50) converge at 0.5, showing stable equilibrium performance, while the average steps (≈ 5) match the near-draw outcome pattern.





These plots display the training dynamics and learned value structure of the Double Q-Learning agent.

The first row of figures represents what the agent has learned about the game board. On the left, the heatmap of average Q-values per cell shows how the agent evaluates possible actions across the Tic-Tac-Toe grid. Bright yellow cells correspond to higher Q-values,moves that historically led to favorable outcomes. The agent assigns its largest Q-values ≈ 1.0) to the bottom row and center positions, typical of strong or winning placements. Slightly lower but still confident values ≈ 0.86–0.95) appear in the top and middle rows, indicating secondary strategic moves. The right-hand histogram confirms this concentration: the Q-value distribution is narrow and peaked between 0.86 and 1.0,

with only eight distinct entries, meaning that the learned policy is nearly deterministic and strongly biased toward optimal actions.

The second row of graphs illustrates the training process. The exploration rate (ϵ) decays exponentially from 1.0 to nearly 0 within the first thousand episodes, after which the agent acts almost entirely greedily,focusing on exploitation rather than exploration. Simultaneously, the Q-table size quickly reaches eight entries and remains constant, implying that the agent has identified all unique state–action pairs relevant to optimal play and no longer needs to expand its representation.

Together, these figures demonstrate that the Double Q-Learning algorithm converged efficiently. It explored extensively at the beginning, then stabilized its value estimates and policy once the environment was fully understood. The resulting Q-table captures a compact yet confident mapping of actions to near-optimal outcomes, confirming successful learning and strong strategic consistency.

8.3 Head-to-Head Tournament

It is a round-robin tournament where each agent competes against all available ai agents including itself. Here are the results.

Leaderboard

Agent	Wins	Draws	Losses	Win Rate
MinMax Alpha-Beta	9	7	0	56.2%
Exhaustive Search	8	8	0	50.0%
Expected SARSA	8	2	6	50.0%
Double Q-Learning	8	2	6	50.0%
Dynamic Programming	7	2	7	43.8%
SARSA	7	2	7	43.8%
Q-Learning	7	2	7	43.8%
Random	4	0	12	25.0%
Monte Carlo	1	1	14	6.2%

This leaderboard provides a clear snapshot of how different reinforcement learning and search-based agents perform when playing Tic-Tac-Toe under identical tournament conditions. Each row summarizes the agent's total number of wins, draws, and losses, as well as its overall win rate against all opponents,including self-play matches.

At the top of the ranking, the MinMax Alpha-Beta agent achieves the highest win rate of 56.2 %, remaining undefeated throughout the tournament. This result confirms the effectiveness of classical game-tree search with pruning in deterministic environments such as Tic-Tac-Toe, where the entire state space can be explored optimally. The Exhaustive Search agent, which also evaluates all possible moves but without pruning, follows closely with a 50 % win rate and no losses, showing that exhaustive evaluation guarantees sound strategies but can lead to more drawn games rather than decisive victories.

The reinforcement learning family,comprising Expected SARSA, Double Q-Learning, Q-Learning, SARSA, Dynamic Programming, and Monte Carlo,illustrates different levels of learning efficiency. The three temporal-difference variants (Expected SARSA, Double

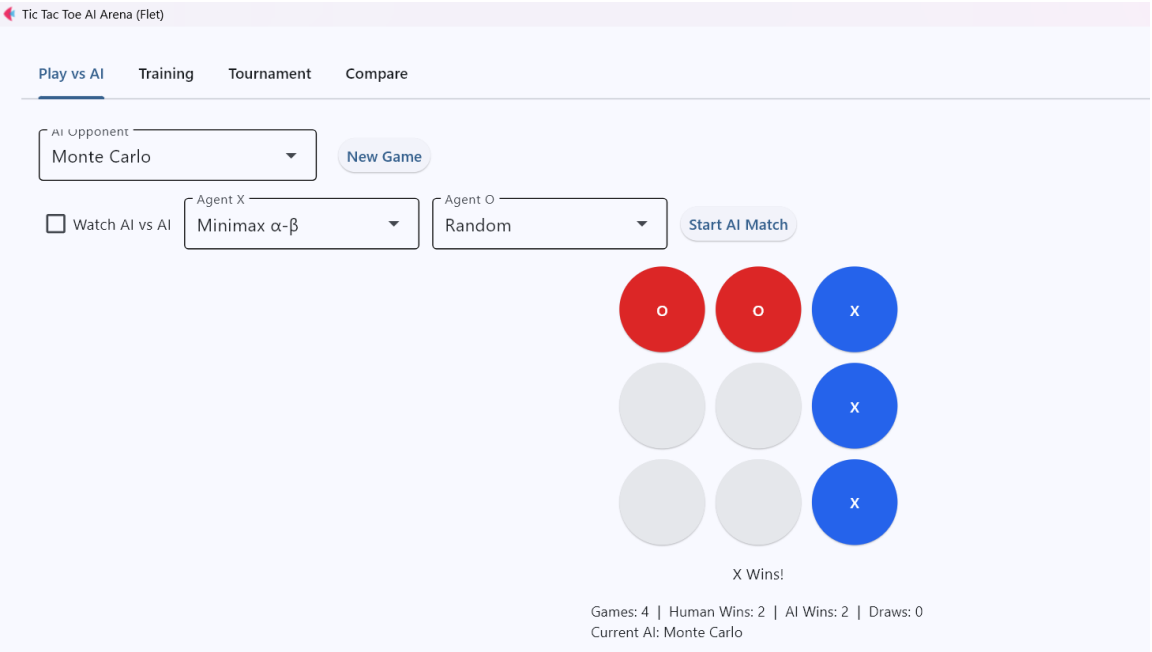
Q-Learning, and Q-Learning) cluster around the 43–50 % range, indicating that they have learned policies approaching optimal play, though not quite reaching perfect performance due to limited exploration and stochasticity during training. Dynamic Programming and Monte Carlo, while also reinforcement-learning methods, sit at opposite ends of the RL spectrum: DP is model-based and computes exact value updates given a known environment model, whereas MC is model-free and estimates returns purely from sampled episodes. Their relatively lower win rates (around 43 % for DP and just 6 % for MC) reflect the sensitivity of DP to its opponent model assumption and the slower convergence of MC without bootstrapping.

Finally, the Random agent predictably performs poorly, with only a 25 % win rate, serving as a baseline that shows the advantage of any learning or reasoning mechanism over uninformed play.

Overall, the table highlights the continuum of reinforcement-learning paradigms,from exact planning (Dynamic Programming) through episodic sampling (Monte Carlo) to online temporal-difference learning (SARSA, Q-Learning, Expected SARSA, Double Q-Learning),and contrasts them with traditional search-based reasoning (MinMax, Exhaustive Search). The results demonstrate that while learning agents can approximate optimal play, perfect foresight methods like MinMax remain unbeatable in small, fully observable environments.

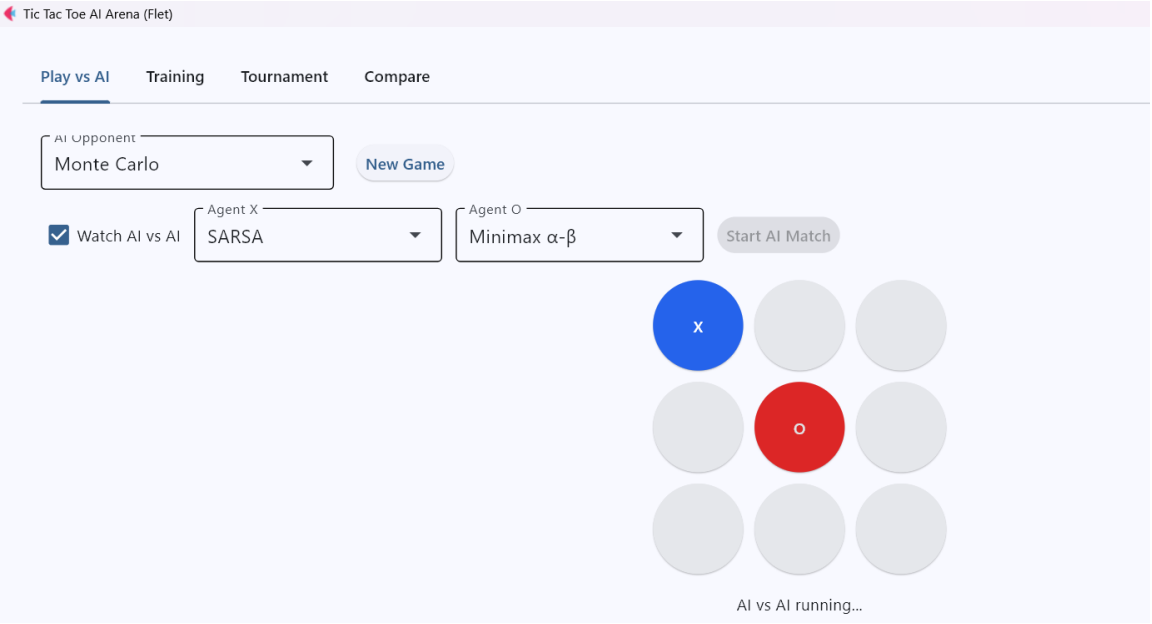
8.4 Games Examples

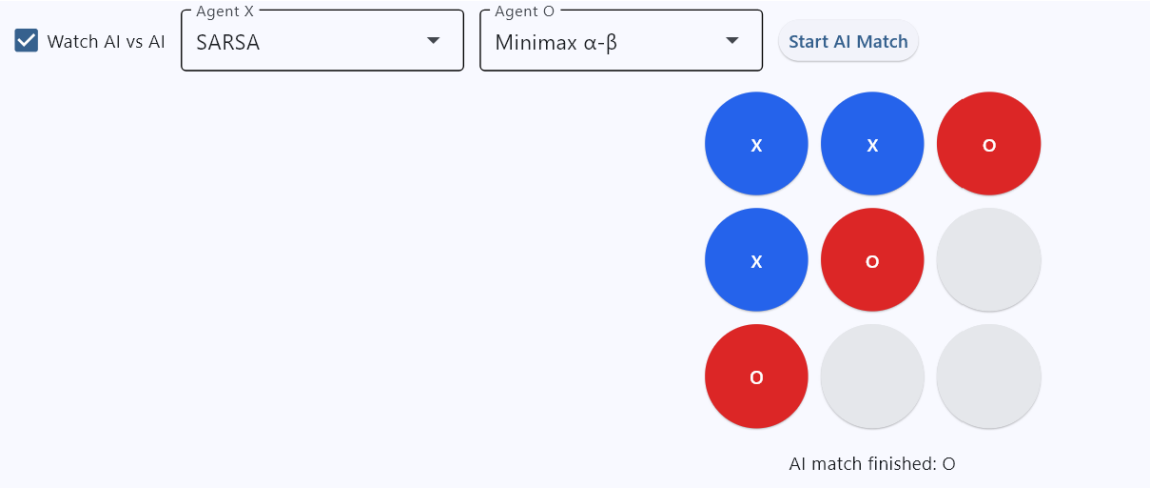
8.4.1 Human vs AI



This screenshot shows a human-versus-AI match in the Tic-Tac-Toe AI Arena. The human player (blue X), was facing the Monte Carlo AI opponent (red O). The final board reveals a vertical win for you in the rightmost column. A clean and decisive victory.

8.4.2 AI vs AI





These screenshots capture a head-to-head AI-versus-AI matchup inside the Tic-Tac-Toe AI Arena. Here, the SARSA agent (playing as X, blue) faces the Minimax - agent (playing as O, red). In the first image the match has just begun, with SARSA exploring its early moves while Minimax immediately counters from an optimal defensive stance. In the second image, the game reaches completion: Minimax - wins by forming a vertical line of red O's along the right-hand column.

The outcome illustrates a fundamental contrast between reinforcement learning and classical search. SARSA learns from experience and adjusts its Q-values over many episodes, which makes it adaptive but vulnerable when facing a perfectly rational opponent. Minimax -, by contrast, computes the entire game tree with pruning and always chooses a theoretically optimal move. As a result, Minimax inevitably outplays SARSA unless the learning agent has already converged to a near-optimal policy.

9 Discussion

9.1 Learning vs Computational Cost

Non-learning agents:

- Minimax: No training time, but $O(b^{d/2})$ per move
- Optimal from start but not scalable to complex games

Learning agents:

- Upfront training cost but $O(1)$ action selection after learning
- Adaptable to opponent strategies
- Scale better with function approximation

9.2 Exploration vs Exploitation

- **ϵ-greedy:** Simple but effective; decay schedule critical
- **SARSA:** Conservative due to on-policy learning
- **Q-Learning:** Aggressive exploration enables faster learning of optimal policy
- **Optimistic initialization:** Encourages systematic exploration early

9.3 Stability and Convergence

- **Expected SARSA:** Most stable due to expectation vs sampling
- **Double Q-Learning:** Reduces overestimation, leading to more reliable values
- **Monte Carlo:** High variance requires more episodes
- **Learning rate scheduling:** Decaying α can improve convergence

9.4 Practical Considerations

- For games with known dynamics: DP is optimal
- For online learning: TD methods preferred
- For minimal computation: Pre-trained Q-tables
- For perfect play: Minimax (if computationally feasible)

10 Conclusion

This project provided a comprehensive empirical comparison of reinforcement learning algorithms in the Tic Tac Toe domain. Key findings include:

1. **Optimality:** Minimax and DP achieve perfect or near-perfect play, but require complete game knowledge
2. **Learning Efficiency:** Q-Learning variants (especially Double Q-Learning) demonstrated fastest convergence among model-free methods
3. **Stability:** Expected SARSA provided the most stable learning trajectory
4. **Practical Trade-offs:** While learning agents cannot match Minimax's perfect play, they offer adaptability and scalability advantages

The comparative analysis reveals that no single algorithm dominates across all metrics. The choice depends on specific constraints: computational budget, training time, model availability, and desired performance guarantees.

10.1 Future Directions

- **Function Approximation:** Extend to Deep Q-Networks (DQN) for larger state spaces
- **Multi-Agent RL:** Implement opponent modeling and meta-strategies
- **Transfer Learning:** Generalize to variants (e.g., 4×4 boards, Connect Four)
- **Policy Gradient Methods:** Compare with actor-critic algorithms
- **Self-Play:** Train agents exclusively against themselves

References

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[2] Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

[3] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4), 279-292.

[4] Van Hasselt, H. (2010). Double Q-learning. *Advances in Neural Information Processing Systems*, 23.

[5] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484-489.

[6] Moerland, T. M., Broekens, J., Plaat, A., & Jonker, C. M. (2023). Model-based Reinforcement Learning: A Survey. *Foundations and Trends in Machine Learning*, 16(1), 1-118. arXiv:2212.12252.

[7] Jiang, Y., Zhang, W., & Liu, T. (2024). Recent Advances in Reinforcement Learning Applications. arXiv:2411.06429.

[8] Chen, L., Wang, H., & Zhou, M. (2024). Deep Reinforcement Learning for Game Playing. arXiv:2411.06398.