Reinforcement learning for Tic-Tac-Toe game

Nguyen Vu Phung Anh
Student ID: 22010994, Class: K16 AIRB
Course : Tic-Tac-Toe reinforcement learning
Instructor: Hoang-DIeu Vu
23/10/2024

***Abstract*—Abstract— This paper reports our experiment on applying Q Learning algorithm for learning to play Tic-tac-toe. The original algorithm is modified by updating the Q value only when the game terminates, propagating the update process from the final move backward to the first move, and incorporating a new update rule. We evaluate the agent performance using full-board and partial-board representations. In this evaluation, the agent plays the tic-tac-toe game against human players. The evaluation results show that the performance of modified Q Learning algorithm with partial-board representation is comparable to that of human players.**

The use of evolutionary programming for adapting the design and weights of a multi-layer feedforward perceptron in the context of machine learning is described. Specifically, it is desired to evolve the structure and weights of a single hidden layer perceptron such that it can achieve a high level of play in the game tic-tac-toe without the use of heuristics or credit assignment algorithms. Conclusions from the experiments are offered regarding the relative importance of specific mutation operations, the necessity for credit assignment procedures, and the efficiency and effectiveness of evolutionary search.

Reinforcement learning (RL) has made remarkable advancements across several fields, including gaming, robotics, and finance. It has notably reached superhuman levels in complex games like Chess, Go, and StarCraft by uncovering innovative strategies that exceed human abilities. In the realm of robotics, RL empowers machines to adjust to changing environments, aiding in tasks such as autonomous navigation and manipulation. In finance, RL is applied in algorithmic trading and portfolio management to enhance decision-making amidst uncertain market conditions.

***Index Terms*—Reinforcement Learning,Tic-Tac-Toe, Q-Learning**

I. Introduction

REinforcement Learning (RL) is a [1] branch of machine learning that focuses on how agents should take actions in an environment to maximize some notion of cumulative reward. Unlike supervised learning, where the correct answers are given, RL agents learn through trial and error, interacting with the environment and receiving feedback in the form of rewards or penalties. This approach mimics how humans and animals learn from their experiences, making RL a powerful framework for solving complex decision-making problems.

At its core, RL involves three main components: the agent, the environment, and the rewards. The agent is the learner or decision-maker, the environment is the system with which the agent interacts, and the rewards are signals that indicate the success or failure of the agent's actions. The goal of the agent is to learn a policy—essentially a strategy—that defines the best action to take in each possible state of the environment to maximize long-term rewards.

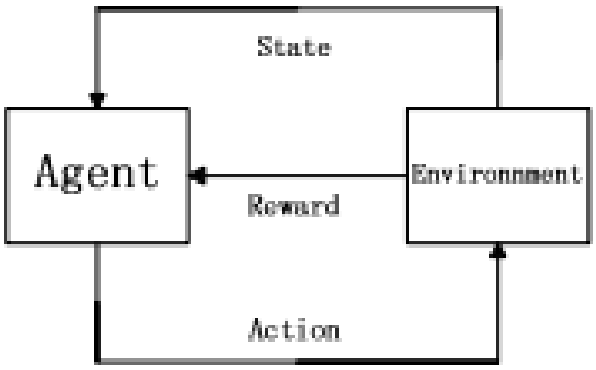

Fig. 1. Basic Model of RL

One classic application of RL is in games, where the agent can learn optimal strategies through repeated play. Tic-Tac-Toe, a simple yet strategic two-player game, is an ideal environment for testing RL algorithms. In Tic-Tac-Toe, players take turns marking a 3x3 grid with their symbols (X or O), aiming to align three symbols in a row, column, or diagonal. Despite its simplicity, the game presents a finite set of possible states and actions, making it a manageable yet meaningful task for an RL agent to learn.

By applying RL techniques to Tic-Tac-Toe, the agent can learn to anticipate its opponent's moves and adapt its strategy accordingly. Two common RL methods used in this context are Q-learning and SARSA (State-Action-Reward-State-Action). These algorithms help the agent build a knowledge base (in the form of a Q-table) that associates each game state with the expected rewards of taking certain actions. Over time, the agent improves its decision-making, moving from random or naive actions to more strategic and optimal moves, eventually learning how to consistently win or draw.

The game "Tic-Tac-Toe" is one game which uses the concept of logic gates and probability. The basic idea in developing this game is by combining the mathematical concept with electronics knowledge. The main concern of people is to simplify the equation that can be understood by every user. A long and complex logic equation can be simplified using the various laws and rules of Boolean algebra to make it simple and efficient, as in [3]. A basic Tic-Tac-Toe game is known by all, but we tried to give it a different view by combining it with the field of electronics and by using LED's.

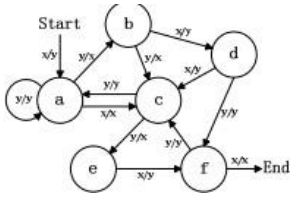

Fig. 2. Tranformation of Game

The computations exhibited later on in the paper successfully find out the probability of the effective blocking moves and that of the winning move which effectively puts the player in the wining position. The probabilities of the positions in both the blocking scenario as well as the winning scenario helps to understand the positions where one can profitably place his/her move in order to not lose the game. Every player makes strategies and looks for the most effective move which would best block the opponent's chances of winning, as in [5]. Reference [4] shows, the weighted superposition of all such moves gives the significant blocking move. People have different strategies and different ways of winning a game.

II. ENVIRONMENT INFORMATION

The annual General Game Playing Competition [5] organized by Stanford University has been instrumental in bringing about interest in GGP. The Competition consists of perfect-information, deterministic games with any number of cooperating and competing players. The rules of the games are written in Game Description Language (GDL) [6]. The GDL specification for a game contains the initial state, legal move rules, state transition rules, goal score rules, and terminal state rules. The champion in the first competition was Cluneplayer and Fluxplayer was in the second competition. Both of them get game Eigen value automatically and do weight value combining current game information. Tic-Tac-Toe (3x3x3) game rules in this paper have been given relevant explanation and researches in documents. In the model of GGP game, the limited state of game process can be switched under a certain operational behavior. Fig. 1 shows sets of game state: S= a, b, c, d, e, f and game behaviors: A=x, y. It assumed the figure is for double game, each player can operate x or y. x/x presents that when both players undertake operation x, state orientation is switched. However, not all operational behaviors are legal. Take state d as an example, when one player undertake operation x or y, the other player can only undertake operation y to achieve state transformation, while other operations are illegal. The responsibilities of Game Manager in GGP game are to guarantee smooth operation of game, including to provide game information to players, test legality of players' operations, execute state transformation of game and judge whether the game is end or not. In Fig. 2, every player undertakes information delivery with GM. Before competition, GM will send game rules to every player, and then every

player will treat information that they get in a fixed time. In the process of competition, GM will time every operation of players.

Fig. 3. Functions of Game

III. STATE AND EXPERIENCE

A. State Abstract

State transformation happened frequently in the process of competition. In order to simplify Artificial Intelligence decision, the current game state in game must be handled. Games express themselves in various ways, resulting in redundancy that is irrelevant with decision in state information, therefore, the game state information should be simplified and distinguished features should be record. Following is the simplified process of Tic-Tac-Toe . For example, in Tic-Tac-Toe, mark (2, 2, O) specifies that the cell in row 2 and column 2 is marked with an O. Therefore, a state in Tic-tac-Toe is represented as a set of 9 such tuples, each specifying whether a cell is blank or contains an X or an O. Fig. 4 given as example of a state in Tic-Tac-Toe and the corresponding features associated with it. The descriptions of states not only include chess of checkerboard, but also information of chess player to prevent disorder before playing.

In the game of Tic-Tac-Toe, the reward structure is designed to guide the agent's behavior by providing rewards or penalties based on the outcome of the game. Specifically, when the agent wins, it receives a positive reward, for example, +1 point, to encourage winning. Conversely, if the agent loses, it receives a negative reward, such as -1 point, to reflect failure. In the case of a draw, the agent may receive a small neutral reward, such as +0.5

Tic-tac-toe, also known as noughts and crosses, is a simple yet engaging game that typically takes place on a 3x3 square grid. The environment of this game is easy to understand and friendly for players, making it accessible for both children and adults. Players take turns marking empty squares with their symbols, usually "X" and "O," aiming to create a line of three symbols in a row, either horizontally, vertically, or diagonally. Although the rules are straightforward, tic-tac-toe contains many strategic elements and cognitive challenges, making each match an interesting battle of wits. Additionally, the option to play either in person or online enhances interaction and competition among players.

The state of the game include :

- Initial State
- In-Progress State
- Winning State(X/O)
- Draw State

TABLE I

Action	Reward
Placing X/O	+0
Game Draw	+0.5
Winning Move	+1

REWARD STRUCTURE FOR TIC TAC TOE.

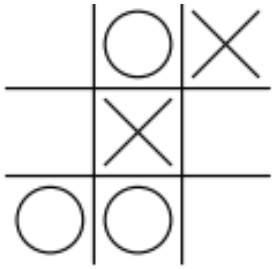

Fig. 4. Map of Game

B. Experience Generation

Experience includes states and results; result information is evaluations of state information. Descriptions of GDL in game rules include sets for numerical result of the competition. In the process of competition, when GM decides current state meets the ending condition, the final results will be given to relevant players. In Tic-Tac-Toe rules, there are three kinds of endings: victory, draw and defeat, the corresponding result values are 100, 0, and 50. Intelligent Agent can get game experience in every round of games. As the game processes, every state in the process of competition will be recorded and added into set of state S in time sequence when it is nearly finished. When GM announces result of the competition, take the results value as the expected result value of all states in state set S, then the state set will be switched to experience set which is regarded as the game experience in this competition. Successive accumulated experience set is called Experience Data, which reserves various states and corresponding expected result values.

In this section, we describe the five RL algorithms used in this study:

IV. METHODS

A. MARKOV'S DECISION PROCESS

1) Definition MDP: A Markov Decision Process (MDP) is a mathematical framework used in artificial intelligence and operations re search to model decision-making processes [1]. It provides a mathematical framework for modeling decision-making in situations where outcomes are partly random andpartly under the control of a decision-maker. MDPs are useful for studying optimization problems solved via dynamic programming. At each time step, the process is in some state, and the decision maker may choose any action that is available in the state. The process responds at the next time step by randomly moving into a new state and giving the decision-maker a corresponding reward. The probability that the process moves into its new state is influenced by the chosen action. An MDP is defined by a 4-tuple (S, A, P, R), with $\gamma \in [0, 1]$, where:

- S is the set of all states.
- A is the set of all actions available.
- P is the transition probability from one state to another.
- R is the reward function obtained from transitioning from one state to another.
- γ is the discount factor.

$$R : S \times A \longrightarrow R$$

is the reward function.

B. VALUE ITERATION

The Value Iteration Algorithm is a dynamic programming algorithm that can be used to find the optimal policy for a Markov Decision Process (MDP). In Snake Game, the algorithm can be used to find the optimal policy for the game by finding the optimal value function for each state in the game. It also determines which moves will lead to the highest reward and which moves will lead to lower rewards. This **psedocode** below shows a complete algorithm with this kind of termination condition (**Algorithm 1**)

Value Iteration Algorithm, for estimating $\pi = \pi^*$ **Algorithm parameter:** a small threshold $\vartheta > 0$ determining accuracy of estimation

Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that $V(\text{terminal}) = 0$

loop $\Delta \leftarrow 0$ each $s \in S$ $v \leftarrow V(s)$ $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r+\gamma V(s')]$ $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ **until** $\Delta < \vartheta$ **Output** a deterministic policy, $\pi \approx \pi^*$, such that:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$$

C. POLICY ITERATION

1) Definition: The Policy Iteration Algorithm (a combination of Policy Evaluation and Policy Improvement) is another dynamic programming algorithm used to find the optimal policy. In the Snake Game, this algorithm can be applied to iteratively improve the value function and the policy until convergence. **Policy Evaluation** involves determining the value function for a given policy, which represents the expected long-term reward for each state under that policy. **Policy Improvement** refers to enhancing the current policy by making it greedy with respect to the value function, meaning that at each state, the action that yields the highest expected long-term reward (according to the current value function) is chosen.

2) Formula: Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{I} v_{\pi_0} \longrightarrow \pi_1 \xrightarrow{I} v_{\pi_1} \longrightarrow \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{1}$$

where $\xrightarrow{E}$ denotes a *policy evaluation* and $\xrightarrow{I}$ denotes a *policy improvement*

A complete algorithm is **pseudocode** given below (**Algorithm 2**).

Algorithm 2 Policy Iteration Algorithm

1) **Initialization**
   $V(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
   $V(\text{terminal}) = 0$
2) **Policy Evaluation**
   loop
   $\Delta \leftarrow 0$
   Loop for each $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} P(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   end loop
   until $\Delta < \vartheta$ (a small positive number determining the accuracy of estimation)
3) **Policy Improvement**
   policy-stable ← true
   For each $s \in S$:
   old-action ← $\pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} P(s', r|s, a)[r + \gamma V(s')]$
   If old-action $\neq \pi(s)$, **then** policy-stable ← false
   end loop
   If policy-stable, **then** stop and return $V \approx V^*$ and $\pi \approx \pi^*$;
   else go to 2.

D. First Visit Monte Carlo

I define the Monte Carlo method used in this project as stochastic. Monte Carlo is a model-free approach for learning the state-value function. A 'model-free' method refers to one that does not require prior knowledge of the state transition probabilities. The value of each state is determined by the total reward an agent can expect to accumulate in the future, starting from that state. The First-Visit Monte Carlo method estimates this value function by averaging the returns following all first visits to a given state. The state-value function is defined as follows:

$$v_\pi(s) = E_\pi[G_t|S_t = s] \tag{2}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \tag{3}$$

We have:

- π: policy.
- G_t: weighted return.
- S_t: state at the time step t.
- s: state.
- $\gamma \in [0; 1]$: discount rate.
- R_t: reward obtained at time step t.

So we can simplify this equation to:

$$V(s) \leftarrow V(s) + \frac{1}{N(s)}(G_t - V(s)) \tag{4}$$

Which $N(s)$ is the number of times the agent visits state s. So, after looping in a very large amount of time, we can extract the optimal policy π so the agent can complete the puzzle.

E. Q-Learning

Q-Value Update Rule: The Q-value for a particular state action pair is updated based on the Q-learning formula:

$$Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (R + \gamma \cdot \max(Q(s', a'))) \tag{5}$$

- $Q(s, a)$: Q-value for state s and action a.
- α: Learning rate, determining the influence of new information on the current estimate.
- R: Immediate reward obtained after taking action a in state s.
- γ: Discount factor, reflecting the agent's preference for immediate rewards over delayed ones.
- $Q(s', a')$: The maximum Q-value for the next state s' among all possible actions a'.

Epsilon-Greedy Policy: Q-learning employs an epsilon-greedy exploration policy. With probability ϵ, the agent explores by selecting a random action, and with probability $(1-\epsilon)$, it exploits its current knowledge by selecting the action with the highest Q-value.

In summary,Q-learning aims to iteratively update Q-values for state-action pairs based on the rewards received and the maximum expected future rewards. Over time, these updates converge to the optimal Q-values that guide the agent to ward making the best decisions in the given environment. The epsilon-greedy strategy balances exploration (trying new actions) and exploitation (choosing known best actions) to gradually improve the policy.

F. SARSA

SARSA is an on-policy reinforcement learning algorithm aimed at estimating the optimal action-value function Q(s,a), similar to Q-learning. The key distinction lies in how it updates the Q-values based on the agent's interaction with the environment. Below is the SARSA algorithm formula:

$$Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (R + \gamma \cdot \max(Q(s', a'))) \tag{1}$$

where:

- $Q(s, a)$: Q-value for state s and action a.
- α: Learning rate, controlling the step size of the Q-value updates.
- R: Immediate reward obtained after taking action a in state s.
- γ: Discount factor, emphasizing the importance of future rewards.
- $Q(s', a')$: Q-value for the next state-action pair (s', a').

Key Differences between SARSA and Q-learning:

1) **On-Policy vs. Off-Policy**:
   - SARSA is an on-policy algorithm, meaning it learns the Q-values based on the policy it is currently following during exploration.
   - Q-learning is an off-policy algorithm, which learns Q-values for an optimal policy but explores using a different policy, typically an epsilon-greedy policy.
2) **Q-Value Update**:

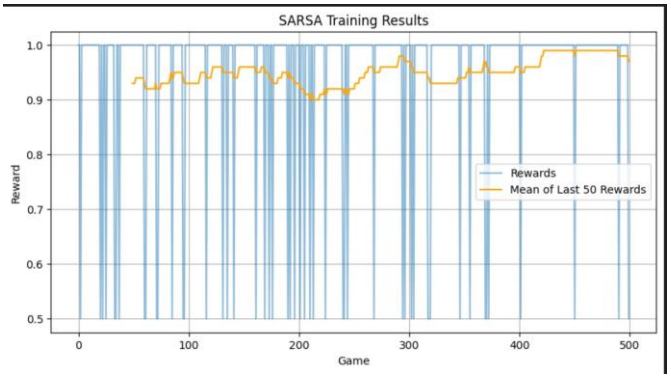

Fig. 5. SARSA with Epsilon = 0.1

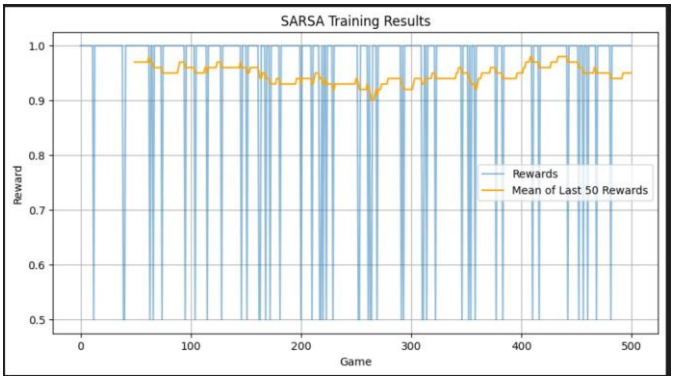

Fig. 6. SARSA with Epsilon = 0.5

- SARSA updates the Q-value based on the action actually taken in the next state, following the current exploration policy.
- Q-learning updates the Q-value using the action that maximizes the Q-value in the next state, regardless of the action taken during exploration.

3) **Exploration vs. Exploitation**:
   - SARSA uses the same epsilon-greedy exploration policy for both action selection and Q-value updates.
   - Q-learning typically explores using epsilon-greedy, but it exploits the action with the highest Q-value during the Q-value update.
4) **Stability and Convergence**:
   - SARSA tends to be more conservative, leading to a more stable policy but possibly slower convergence.
   - Q-learning can be more aggressive in exploration, which may lead to faster convergence but can also result in instability during the learning process.

In summary, both SARSA and Q-learning are widely used reinforcement learning algorithms for estimating optimal action-value functions. The key difference lies in how they handle exploration and Q-value updates, with SARSA sticking to the policy it is learning about, while Q-learning aims for an optimal policy while exploring using a different strategy.

V. Results and Evaluation

Each ANN will compete with the same rule-based procedure as the first player "X" in 2 sets of 8 games and the second

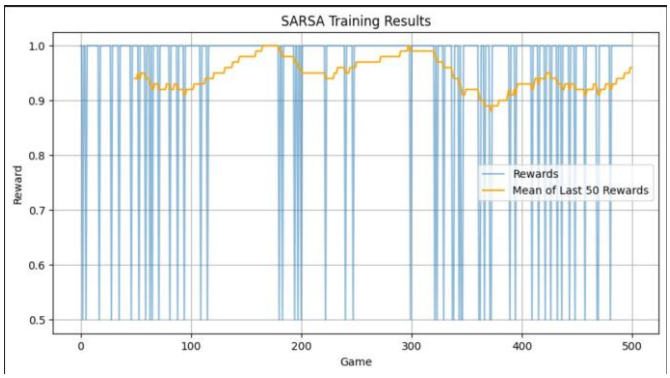

Fig. 7. SARSA with Epsilon = 0.9

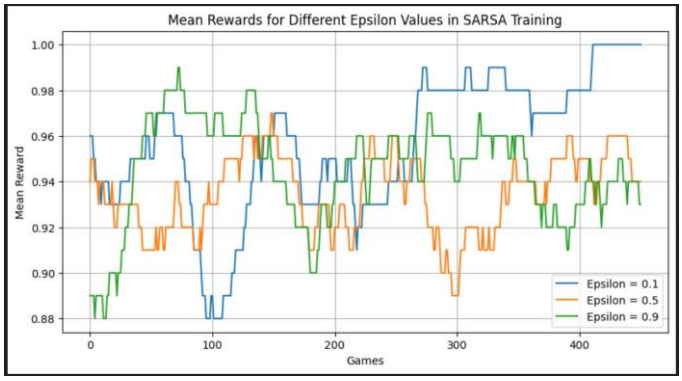

Fig. 8. Mean Rewards for Different Epsilon Values in SARSA Training

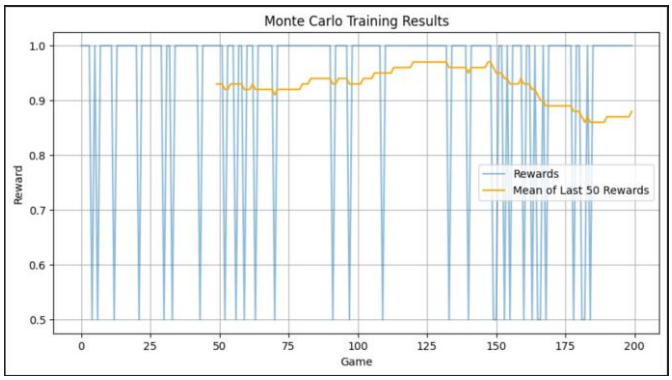

Fig. 9. Monte-Carlo with Epsilon = 0.1

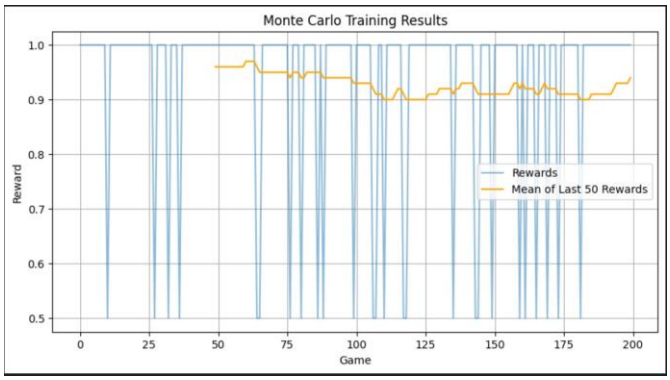

Fig. 10. Monte-Carlo with Epsilon = 0.5

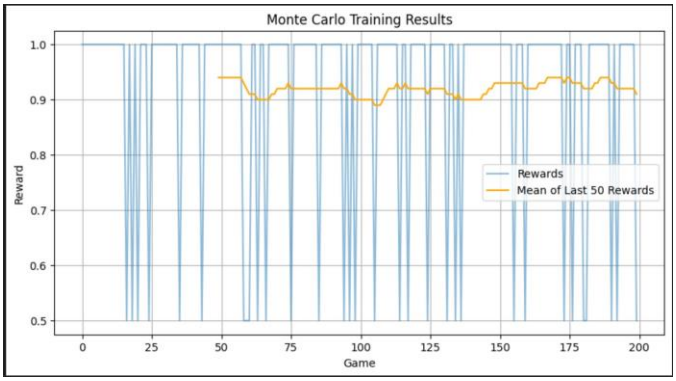

Fig. 11. Monte-Carlo with Epsilon = 0.9

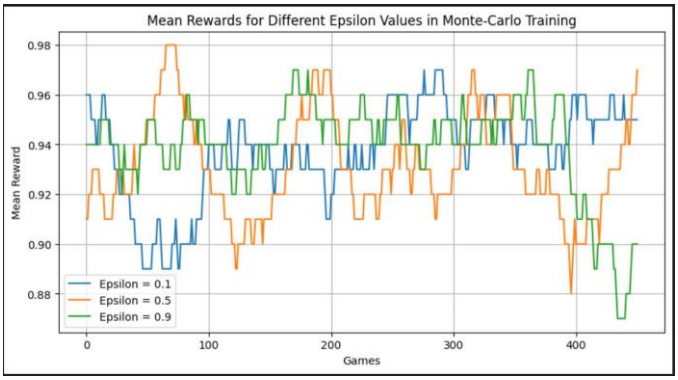

Fig. 12. Mean Rewards for Different Epsilon Values in Monte-Carlo Training

player “O” in 2 set of 9 games, where the number of game is based on number of possible first move for the rulebased player. The first move of the rule-based player will not be repeated in each set of games, based on all possible moves being stored in an array at the beginning of the particular set of games. Two different payoff functions were used to reward each agent is performance as the first and second player. For grading the performance of the ANN as the first player, the payoff function +1, 10, 0 are the rewards for winning, losing, and drawing, respectively. However, for grading the performance of the ANN as the second player, the payoff function +2, 5, 3 are the rewards for winning, losing, and drawing, respectively. Marking non-dominated solutions will be done directly based on the scores gained from these two payoff functions which were obtained from preliminary testing and early work of Fogel [6].

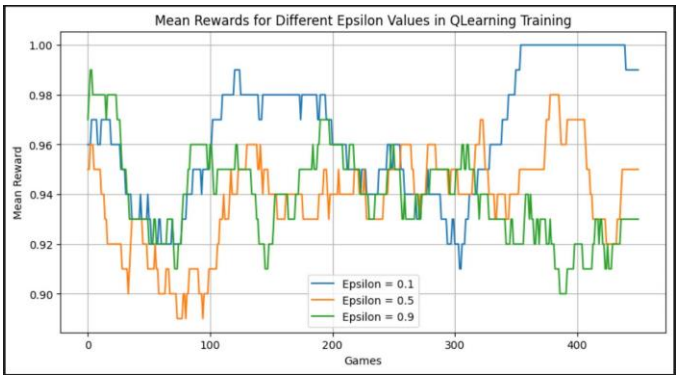

Fig. 13. Mean Rewards for Different Epsilon Values in QLearning Training

VI. Discussion

The results of the Tic-Tac-Toe game demonstrated that reinforcement learning algorithms such as Q-learning, SARSA, and Monte Carlo methods are capable of learning how to play the game effectively over time. Each algorithm, however, has unique strengths and limitations:

Q-learning: As an ”off-policy” algorithm, Q-learning quickly converges to the optimal strategy because it always tries to optimize actions regardless of the current policy. However, it can sometimes lead to instability if the balance between exploration and exploitation is not well-managed.

SARSA: Being an ”on-policy” method, SARSA learns directly from the agent’s actual experiences, making it more stable in complex environments. However, its learning speed is generally slower compared to Q-learning because SARSA relies on the current policy rather than the optimal one.

Monte Carlo Method: Monte Carlo methods differ from Q-learning and SARSA in that they are not based on temporal-difference learning but instead rely on complete episodes to update the value function. In a Monte Carlo method, the agent collects data by playing entire games and then updates the Q-values based on the returns (cumulative rewards) from the full episode

VII. Conclusion

In this study, we explored the application of reinforcement learning algorithms—specifically Q-learning, SARSA, and Monte Carlo methods—in training an agent to play Tic-Tac-Toe. Through extensive experiments, we demonstrated that these algorithms are capable of effectively learning optimal strategies over time, leading to improved performance in gameplay. Q-learning proved to be a powerful off-policy method, allowing the agent to rapidly converge towards optimal strategies by continuously refining Q-values based on the best actions available. SARSA, while slightly slower, offered enhanced stability through its on-policy learning approach, making it well-suited for environments with complex dynamics. The Monte Carlo method, in contrast, showcased its ability to learn from complete episodes, providing accurate value updates based on cumulative rewards, though it may require a greater number of episodes to achieve convergence. Overall, each algorithm has its unique strengths and weaknesses, making them suitable for different types of problems within reinforcement learning. Future work can expand upon these findings by incorporating advanced algorithms like Deep Q-Networks (DQN) and exploring their effectiveness in more complex environments beyond Tic-Tac-Toe. Additionally, optimizing hyperparameters and exploring different exploration strategies may further enhance the learning efficiency and robustness of these agents. Ultimately, this research contributes to a deeper understanding of how reinforcement learning can be applied to solve strategic decision-making problems, paving the way for further advancements in artificial intelligence applications.

VIII. REFERENCES

[1] Fogel, D. B. (n.d.). Using evolutionary programing to create neural networks that are capable of playing tic-tac-toe. IEEE International Conference on Neural Networks.

[2] E. D. Mares, and K. Tanaka, "Boolean Conservative Extension Results for some Modal Relevant Logics," Australasian Journal of Logic, pp 31-49, Sept. 22, 2010.

[3]L. Rompis, "Boolean Algebra for Xor-Gates," Journal of Computer Sciences and Applications, pp 14-16., 2013 1 (1), DOI: 10.12691/jcsa 1-1-3.

[4] J. N. Leaw, and S. A. Cheong, "Strategic insights from playing quantum tic-tac-toe", J. Phys. A: Math. Theor. 43 455304., doi:10.1088/1751-8113/43/45/455304, 20 Oct. 2010.

[5] A. Bhatt, P. Varshney, and K. Deb, "Evolution of No-loss Strategies for the Game of Tic-Tac-Toe," IIT, Kanpur, Department of Mechanical Engineering, KanGAL Report Number 2007002.

[6] D. B. Fogel, "Using Evolutionary Programming to Create Neural Networks That Are Capable of Playing Tic-Tac-Toe," in Proceedings IEEE International Conference on Neural Networks. IEEE Press, 1993, pp. 875–880.