Comparative Study of MCTS, Q-Learning, and SARSA in the Game of Chess

https://github.com/hyk029/COGS-188-FP.git

Group Members

Han Young Kim

Adrian Zhu Chou

Veeva Gathani

Abstract

This project investigates how hyperparameter tuning influences the performance of reinforcement learning algorithms for the game of chess. We focused on Monte Carlo Tree Search and temporal difference methods such as Q-Learning and SARSA. We address the enormous state space by integrating feature extraction and neural networks within a custom ‘python-chess’ and OpenAI Gym environment. A randomly generated chess dataset, containing FEN strings, move sequences, and game outcomes, provide the training and evaluation data. Key hyperparameters (learning rate, discount factor, exploration rate, tree search depth) are tuned using random search to ensure consistent and unbiased comparisons across all three methods. Performance is measured in terms of convergence speed, win rate against baseline heuristic agents, and efficiency.

Background

Reinforcement learning has undergone a huge development and has shown immense potential in problem-solving and decision making abilities. It is the closest thing to the learning used by humans and animals. It has been known for its success across several domains of robotics, autonomous agents, etc. It includes mapping situations to actions so that a numerical reward signal is maximized [1]. A notable example of successful reinforcement learning used in chess was in 1997 when IBM's Deep Blue defeated World Champion Garry Kasparov. It had a combination of vast computational power and an extensive database of opening moves and endgame strategies [2].

Chess was at the forefront of artificial intelligence research for much of the first 50 years of the field. It was and is used by AI researchers because it is a game of long-term planning and predicting the moves of your adversary. Being able to predict what would happen in 50000 moves from now with a balance of risk and reward is quite difficult for humans [3].

When developing Deep Blue, machine learning methods for game-playing programs were fairly primitive and didn't help much. Algorithms for efficient search and evaluation of possible combinations were utilized instead. Since then, machines have improved in processing speed and memory [4]. More recent work includes SARSA and Q-Learning being applied to chess and other board games looking at how learning rate and discount factor affect performance [5].

Despite all this background, a comparative analysis of how hyperparameter choices affect different reinforcement learning approaches in chess is underexplored. Most of the past research has been focused on winning games and reaching peak performance instead of evaluating how various parameters influence learning stability and computational cost. Hence, this lack of exploration prompted our research question and by tuning hyperparameters across Q-learning, MCTS, and SARSA, we aim to understand the effects of such adjustments.

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd Edition)*. MIT Press. Retrieved from https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
[2]Whitehead, R. (2023). *How Machine Learning and Reinforcement Learning Have Transformed Chess and Strategic Games***.** IOA Global. Retrieved from https://ioaglobal.org/blog/how-machine-learning-and-reinforcement-learning-have-transformed-chess-and-strategic-games/
[3] Thompson, T. (2023). *AI in Chess: How Chess AI Works and What It Means for the Future of the Game*. modl.ai. Retrieved from https://modl.ai/chess/
[4] Greenemeier, L. (2017). *20 Years after Deep Blue: How AI Has Advanced Since Conquering Chess.* Scientific American. Retrieved from https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess/
[5] Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and reinforcement learning in board games. In *Handbook of game theory with economic applications* (Vol. 4, pp. 885-895). Elsevier.

Problem Statement

The performance of reinforcement learning varies depending on hyperparameter settings, yet there is limited systematic research examining how these parameters influence behavior and efficiency in complex domains like the game of chess. Chess offers an enormous state space, a vast action space, and nuanced dynamics, making it a particularly challenging environment. This

problem is quantifiable in that performance can be evaluated through metrics such as win rate, convergence speed, and efficiency (computational cost). It is measurable via controlled experiments that isolate specific hyperparameters of interest (learning rate, discount factor, exploration strategy, tree search depth) under consistent conditions across all methods and data inputs. Because chess engines, datasets, and RL methods are publicly available and standardized, the problem is replicable across different implementations, system setups, and subsets of chess positions. By systematically exploring hyperparameter tuning within various reinforcement learning algorithms, (Monte Carlo Tree Search, Q-Learning, and SARSA) this problem is replicable across different chess positions/phases, game scenarios and algorithm implementations, making findings generalizable and applicable to similar complex strategic environments.

Data

For this comparative study of reinforcement learning algorithms in chess, we used two approaches for our datasets: standard chess starting positions and custom randomly generated positions.

Standard Chess Starting Position:

For the standard chess starting positions dataset, we used the ChessEnv class. This environment generated data internally by using the standard chess starting positions and allowing the agents to explore the game through self-play. This allowed us to obtain a consistent baseline for algorithm comparison without needing to acquire external datasets.

Custom Random Generated Positions:

For the custom random generated positions dataset, we used the FENDatasetChessEnv class. This environment allowed us to load the chess positions in Forsyth-Edwards Notation (FEN) from CSV files. We created a tool (create_dataset.py) that allowed us to test the algorithms on different board states other than the standard starting positions.

This tool is capable of:

- Generating random positions by playing random legal moves from the initial positions.
- Extracting the positions from Portable Game Notation (PGN) files of recorded chess games.
- Saving positions to CSV format with metadata.
- Filtering for positions with balanced material.

We created a small test dataset (chess_positions.csv) that contains 76 positions with the following structure:

- Fen: String representation of the chess position.
- Balanced: Boolean indicating whether material is balanced.
- Halfmoves: Integer count of halfmoves since the last capture or pawn advance.

Critical Variables:

- Board State: Represented as a 64 element integer array with values ranging from -6 to 6, where positive values represent white pieces (1 to 6) and negative values represent black pieces (-1 to -6), with the absolute value representing the piece type.
- Action Space: Encoded as integers from 0 to 4095, representing the possible moves from any square to any other square (64 x 64 = 4096 possible moves).
- Material Balance: For positions in the custom dataset, we track whether the material difference between players falls within a predefined threshold (default: 2 pawns).

Memory Usage and Computational Statistics:

- Q-Learning and SARSA: The Q-table typically grew to 10,000 - 50,000 entries by the end of the training.
- MCTS: The search tree was limited to 5,000 nodes to control memory usage, with pruning implemented to maintain the most valuable nodes.

Challenges:

We ran into problems when trying to use larger datasets for training. The complexity of chess makes processing large position datasets resource intensive. When testing the larger external datasets, our environment experienced performance issues and crashes. This was due to:

- Memory limitations when loading and processing large numbers of positions.
- Computational overhead of the MCTS algorithm with large datasets.
- Growth of the Q-table in Q-Learning and SARSA implementations.

Therefore, for the final analysis, we primarily used the built-in environment with standard starting positions, as well as random generated positions. This allowed sufficient data for the algorithms to be compared while maintaining computational efficiency.

Proposed Solution

Approach:
We propose a comparative hyperparameter tuning paper using:

1. Monte Carlo Tree Search (MCTS)
2. Q-Learning
3. SARSA

These algorithms will interact with a chess environment built using python-chess for game logic and OpenAI Gym for a standardized RL interface.

Key hyperparameters:

1. Learning rate: step size in value updates
2. Discount factor: importance of future rewards
3. Exploration rate: exploration-exploitation
4. Tree search depth (for MCTS): how deep simulations go

Process:

1. Create a custom chess environment for board representation
2. Link the environment inside OpenAI Gym so that it is exposed to step(action), reset() and legal_moves.
3. Define the reward function (how many points are awarded or deducted based on win, lose, draw)
4. Train all algorithms using default hyperparameters to create a performance benchmark.
    - This would include initializing Q-table mapping to Q-values (each state-action pair starts with an initial Q-value), and search trees (each node tracks visit counts and average rewards).
5. Run training episodes where the board is reset to initial state. Legal moves and current state representation is noted. Select and apply moves using algorithm policy. Observe the next state and reward and update agents.
6. Record game results based on total moves, times per move, and final outcome (win/loss/draw).

Evaluation Metrics

Convergence Speed
Measured by the number of episodes required to reach a performance threshold T. Mathematically, if Wi is the performance at episode i, we define convergence speed as the smallest i such that $W_i \geq T$

Computational Efficiency
Assessed through training time and resource utilization (CPU/GPU usage) needed to train the model until convergence. Formally, if ti is the total training time up to episode i, then computational efficiency can be reported as ti at convergence.

Performance
Performance will be measured with the final win rate against heuristic agents. If W is the number of wins and G is the total number of games, then the performance is given by W / G.

Results

This section will discuss the results we obtained from comparing the performance of MCTS (Monte Carlo Tree Search), Q-learning, and SARSA algorithms when playing chess under different hyperparameter settings. As mentioned before, we evaluate the performance of these algorithms based on convergence speed, computational efficiency, performance, and win rate against heuristic agents.

Baseline Performance

Before we tuned any hyperparameter settings, we wanted to check the default hyperparameters and the performance against heuristic agents.

Figure 1 indicates the default performance. As shown, none of the reinforcement learning algorithms won any of the games in the default settings. We know this as the win rate is 0% for all. It constantly remained at 0% throughout for MCTS, SARSA, and Q-Learning, highlighting how it failed to learn anything to improve performance. Additionally, no policy learning was noticed either, indicating an imbalance in exploration and exploitation. Hence, it emphasizes the importance of tuning hyperparameters to achieve good performance and learning.

Base Model Performance and Learning Curve

Once we had chosen our dataset, and configured it based on our necessity, we evaluated the initial training performance of each model over time and episodes based on several hyperparameters.

Figure 2, 3, and 4 highlight the performance metrics and learning curves of MCTS, Q-Learning, and SARSA over learning episodes respectively.

As seen in **Figure 2**, MCTS's graphs hardly had any changes in graphs and performance. The lack of drastic changes in reward values could indicate that the model failed to develop a proper and useful strategy within the episodes it trained for. It also shows that over time, for each episode, the episode length increased linearly, indicating that whilst it took longer, it didn't improve. Furthermore, the lack of cumulative checkmates highlights the inability to reach a winning stance against the agents. Lastly, the memory usage was the highest towards the end compared to SARSA and Q-Learning. This indicates MCTS did consume the most resources when being trained.

As seen in **Figure 3,** Q-Learning's rewards graph shows a sudden drop in total reward just after episode 20. This could represent a new policy shift, which didn't work out. There was no consistency in the episode lengths, but the illegal moves per episode did decrease exponentially

over time and episodes, showing how the model did eventually learn valid moves in chess. Furthermore, an interesting graph is the one that shows the cumulative checkmates achieved by the model over episodes. As shown, just after episode 20, the model came across a drastic shift (most probably due to the policy shift) and was able to produce one checkmate, showing some progress.

Lastly, as shown by **Figure 4,** SARSA's rewards graph shows that the rewards remained consistently at 0, representing minimal learning progress. Moreover, episode lengths fluctuated significantly, which could indicate unbalanced and unstable policy learning. Like the MCTS model, the SARSA model also failed to reach any winning strategies as the cumulative checkmates remained at zero throughout.

Hence, in comparison, the Q-Learning model performed the best compared to SARSA and MCTS showing minimal improvements in policy implementation and reduced illegal moves. But, as all did struggle to win, more hyperparameter tuning is required.

Hyperparameter Tuning + Model Selection

To drastically improve performance and model learning, we tuned the learning rate, discount factor, and exploration rate.

As **Figures 5** and **6** show, final average rewards varied significantly with different alpha values. The best performing values were between 0.05 and 0.1, where the reward was highest. But, for the q-learning model, the alpha values between 0.4 and 0.5 also indicated high reward. As seen in **Figure 5,** alpha value 0.2 represents a sudden drop in performance, which could mean unstable policy updates. Additionally, memory usage peaked at alpha 0.01 but dropped at 0.2 which could indicate that higher alpha values led to more policy overwrites than learning.

As seen in **Figure 6,** similarly, SARSA performed best at 0.05-0.1 alpha values; but, it declined a lot at alpha 0.2. Moreover, memory usage peaked at alpha = 0.1 but was inconsistently reducing after that. In conclusion, it seems that the Q-learning model handled learning rate variations better than SARSA.

Figure 7 and **8** show the impact of epsilon decay on Q-learning and SARSA respectively. It shows how gradual epsilon decay (from 0.995 to 0.999) allowed for better exploration early on which it aimed to stabilize later. A fixed epsilon didn't work as well. It led to premature exploitation which reduced the overall win rate.

Lastly, **Figures 9** and **10** indicate the impact of the discount factor on Q-learning and SARSA models. The optimal discount factor for the best balance between immediate and long-term rewards was 0.95. A discount factor of 0.99 shows a decrease in learning speed as agents potentially aim for distant rewards instead, making the process slower.

Figure 11 shows the MCTS performance versus the number of simulations. It shows how increasing simulations improved win rate but had worse returns when a threshold was crossed. Additionally, too many simulations significantly increased computational time per move.

Model Comparison

Overall, based on **Figures 12**, we can see that MCTS achieved the highest win rate after tuning but of course, it required the most computational resources. Conversely, Q-Learning and SARSA significantly improved and were more efficient.

Discussion

Interpreting the result

The key finding from our experiments is that careful tuning of hyperparameters (learning rate, discount factor, and exploration rate) substantially affects each algorithm's performance in the game of chess. Across our three methods, optimal hyperparameter sets led to faster computing time and higher win rates. The opposite is true. Something of note was that methods with dynamically adjusted exploration strategies (decaying epsilon in an epsilon greedy strategy) tended to stabilize performance in a very effective manner. Essentially, correctly tuning and selecting parameters was crucial for us, especially due to limiting hardware capabilities.

Delving deeper, MCTS showed admirable performance in its ability to handle branching factors in middle game positions/phases but required substantial computational resources when the depth of the tree search increased. We especially remember how frustrating it was when the algorithm crashed over night or how it would take ages to even complete episodes and simulations. This performance however allowed MCTS to outperform the tabular methods in complex positions due to it giving priority to the most promising branches. A big trade off was time however. We eventually realized that once a certain node limit was reached, deeper tree searches only offered minimal upsides which made it not worth it, especially due to its demanding nature and the limiting computational power at our disposal. It was basically a war between robust performance and efficiency.

On the other hand, Q-Learning and SARSA both had significantly smaller computing costs than MCTS. This came at a cost however, because they were more susceptible to bad hyperparameter configurations. We noticed that when the learning rate was too high, both algorithms over-adjusted and went back and forth between suboptimal policies. To mitigate this, we noticed that a more conservative learning rate fixed this issue with the trade off of a slower convergence rate. This finding highlights the importance of really balancing exploration and learning rates in convergence. The former is especially true for larger state spaces such as the game of chess.

An interesting thing we noted was that it was helpful to look into different types of phases or positions in the game (rather than just the opening phase/positions) during training because this yielded more solid performances across various phases of chess (beginning, middle, end) because it 'forced' the algorithm to learn strategies when being put on the spot in other situations, thus helping reduce overfitting and making the algorithm more adaptable.

Limitations

One limitation of our study is that larger datasets were not extensively tested due to memory and processing constraints: the Q-table could inflate dramatically, and MCTS struggled with time when exploring too many positions/phases. Another issue is that we focused on a relatively narrow set of hyperparameters. Additionally, our experiments relied on straightforward reward signals (win, loss, draw), so exploring more nuanced reward shaping could have revealed different behaviors or accelerated learning. Lastly, hardware constraints prevented us from running very long training sessions, which may mask long-term trends in performance and convergence.

Ethics & Privacy

While the chess datasets used are publicly available, ethical considerations remain relevant when developing decision systems that are automated. Automated decision-making in competitive environments must be transparent, fair, and explainable. Our team will address potential biases and unintended consequences by ensuring model explainability, rigorous testing, and adherence to responsible AI guidelines.

Some ethical concerns could include:

- Bias in training data: if the datasets are selectively featuring specific levels of players and games, or biased against race, demographic, etc, the training agents would also be compromised.
- Exploitation: if the agents are only tuned to maximize win rate, they might find "unfair" strategies or cheating schemes exploiting the environment instead of measuring actual skill and performance.
- Overfitting might also occur if the hyperparameter tuning is over-optimized, leading to the agents not being able to effectively perform against diverse opponents.

Hence, in order to ensure that these ethical considerations are kept in check, we need to ensure the datasets we use are diverse and representative, covering all games across all skills and types. Additionally, we would potentially consider conducting tests against baseline agents to evaluate generality. Lastly, we will make sure to follow all AI guidelines to check for ethical blindspots.

Conclusion

This study provides insight into how hyperparameter tuning influences the performance of Reinforcement Learning algorithms in the game of chess. Our comparative analysis of Monte Carlo Tree Search, Q-Learning and SARSA algorithms revealed performance variations based on hyperparameter choices.

Our results demonstrate that each algorithm responds differently to parameter adjustments, with MCTS exhibiting admirable and robust performance in complex positions but requiring more computational resources. On the contrary, Q-Learning and SARSA offered computational efficiency at the cost of being more sensitive to parameter configurations.

These findings contribute to the development and understanding of Reinforcement Learning research by highlighting the importance of systematic hyperparameter tuning in complex strategic environments. While chess requires extensive training and computational power, our work was still able to show that carefully tuned traditional reinforcement learning methods can develop meaningful chess strategies.

Future Work

A logical next step would be to incorporate larger and more diverse position datasets, potentially leveraging distributed training to manage memory usage. Fine-tuning or dynamic scheduling of hyperparameters (for instance, automatically adjusting tree search depth based on board complexity) could improve efficiency when it comes to MCTS. Incorporating function approximation via deep neural networks in place of a tabular Q-table or shallow search heuristics could also enable learning more nuanced state representations. Finally, adapting reward structures to consider piece values or positional heuristics might offer better feedback and be more representative of actual chess heuristics, thus improving training stability and depth of learning.

Appendix

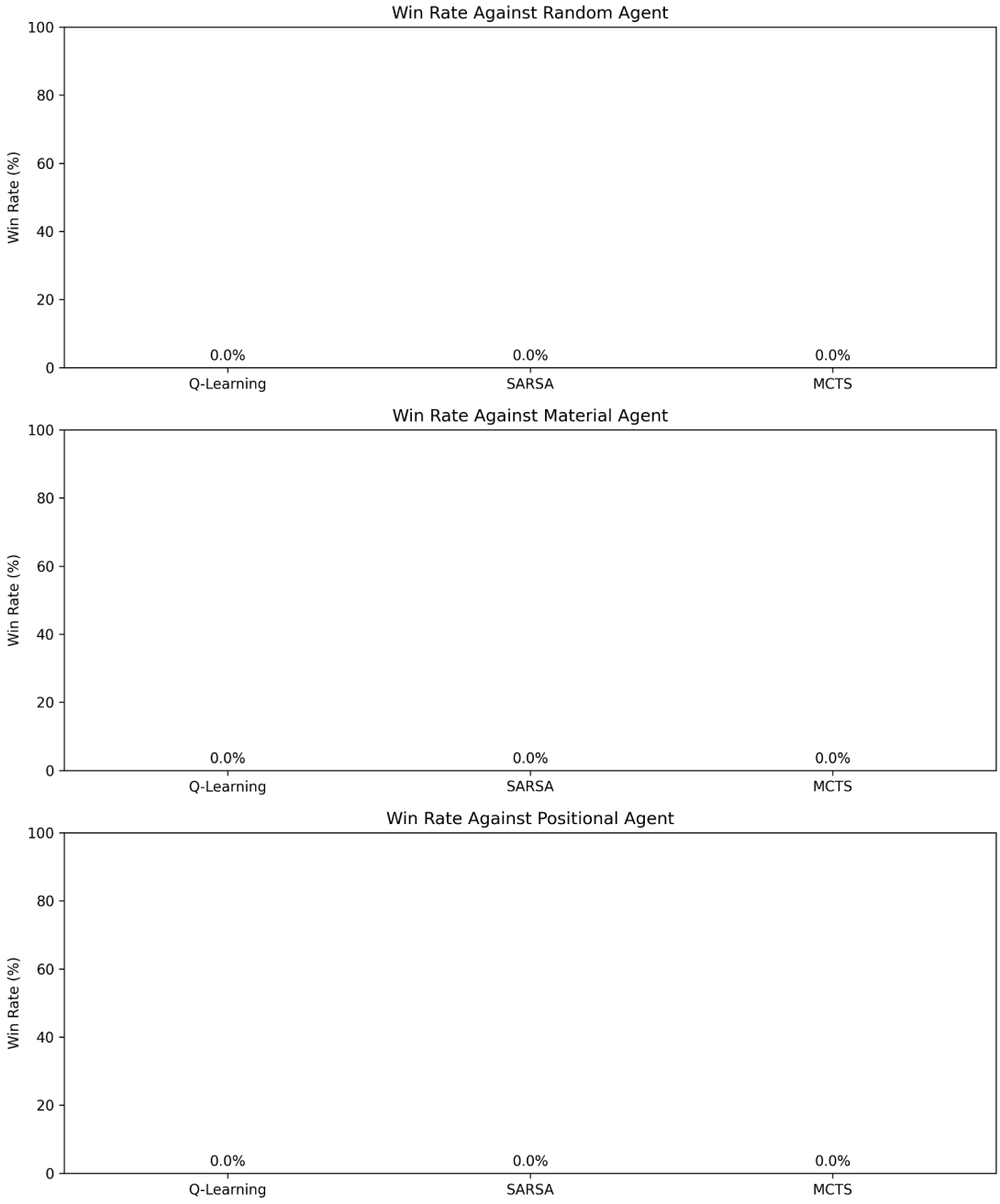

Figure 1: Baseline comparison across MCTS, SARSA, and Q-Learning algorithms

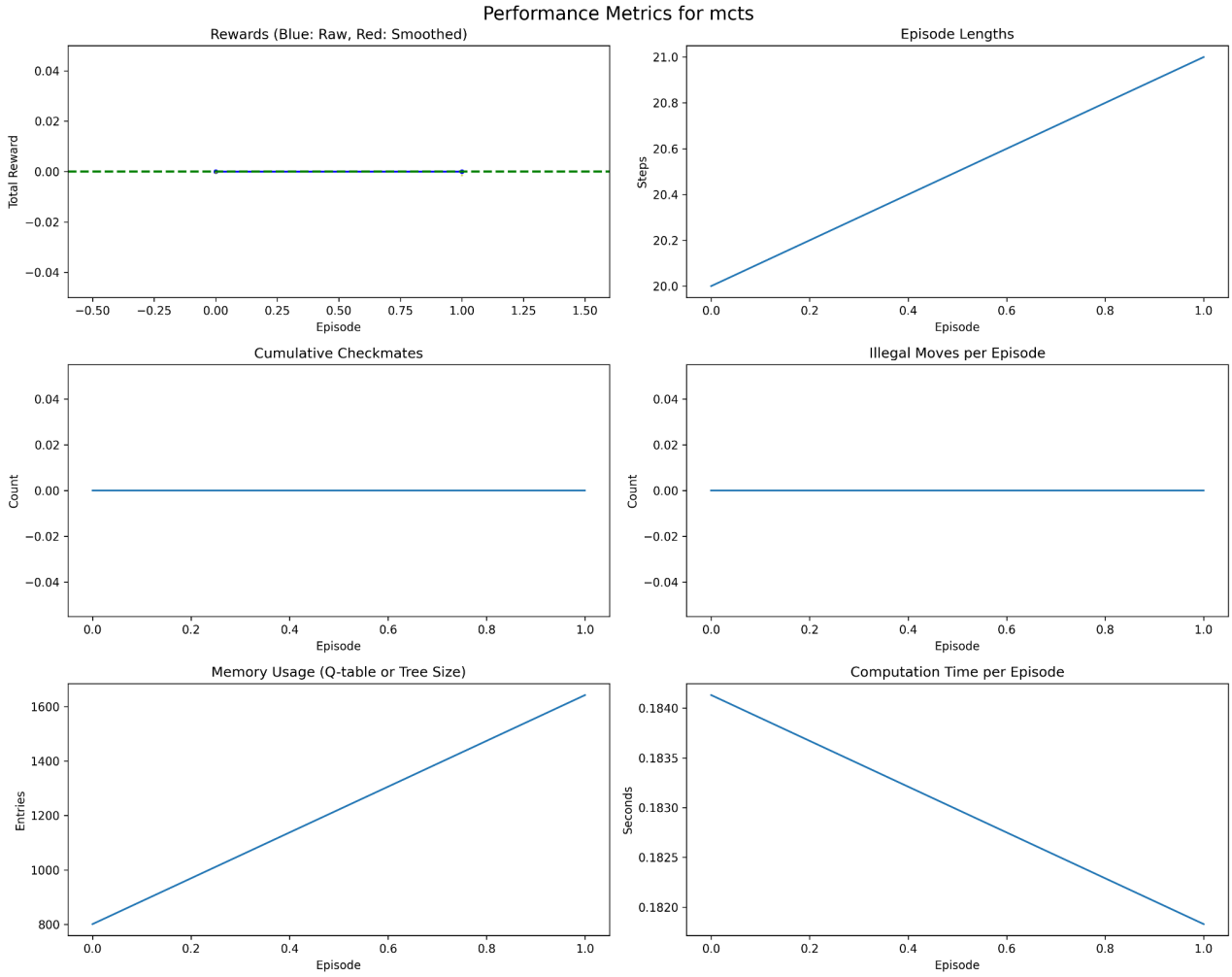

Figure 2: Learning Curve of MCTS over training episodes

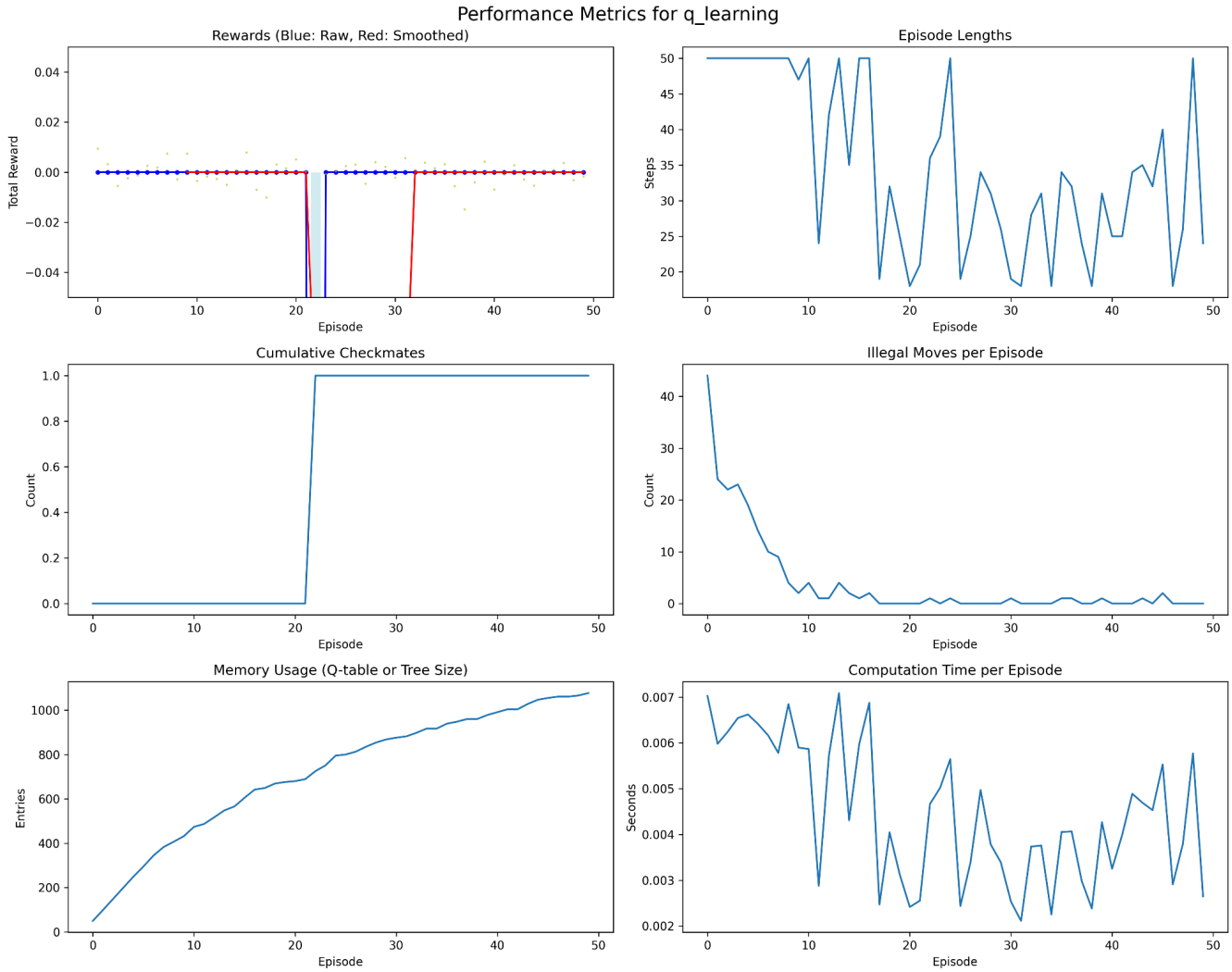

Figure 3: Learning Curve of Q-Learning over training episodes

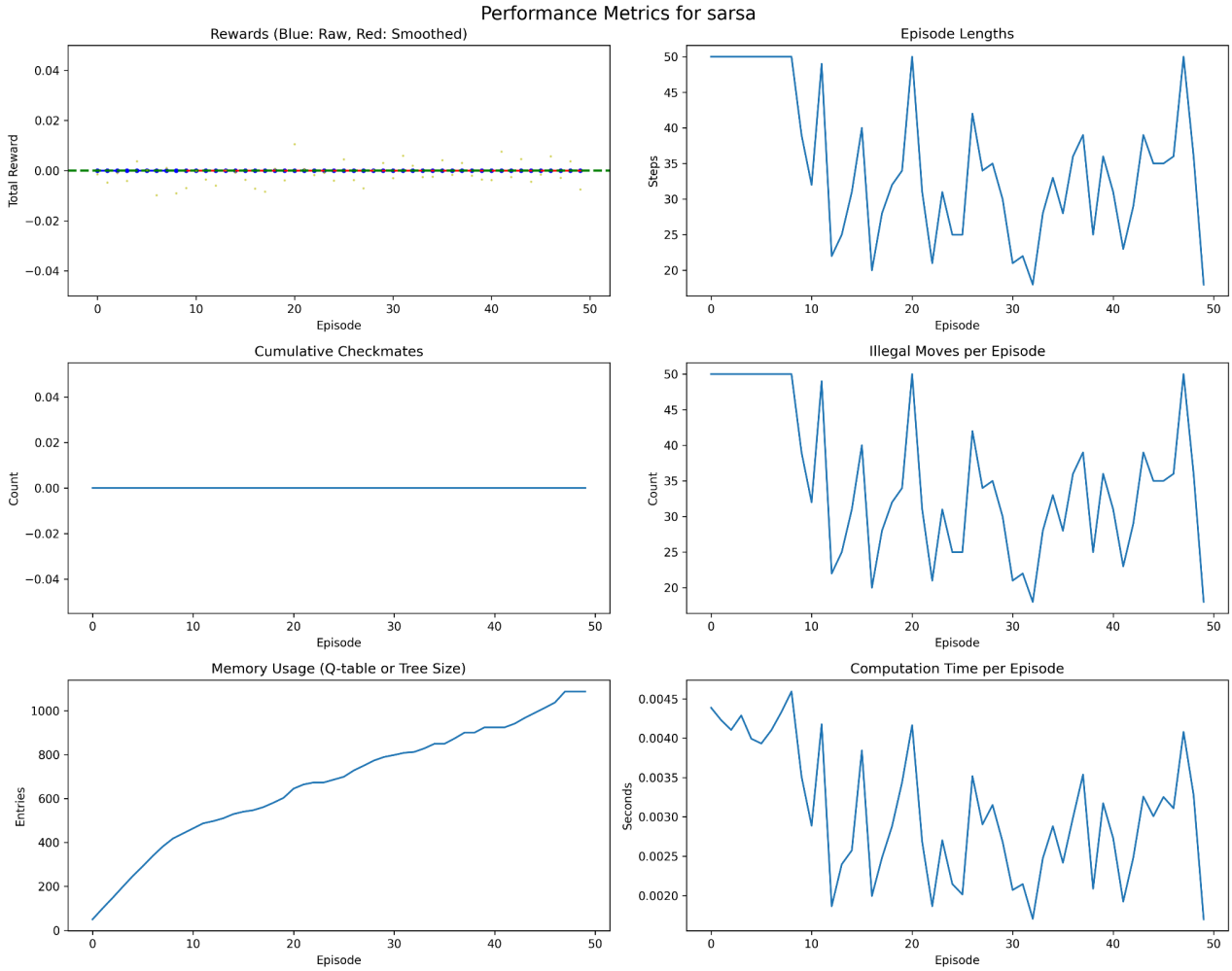

Figure 4: Learning Curve of SARSA over training episodes

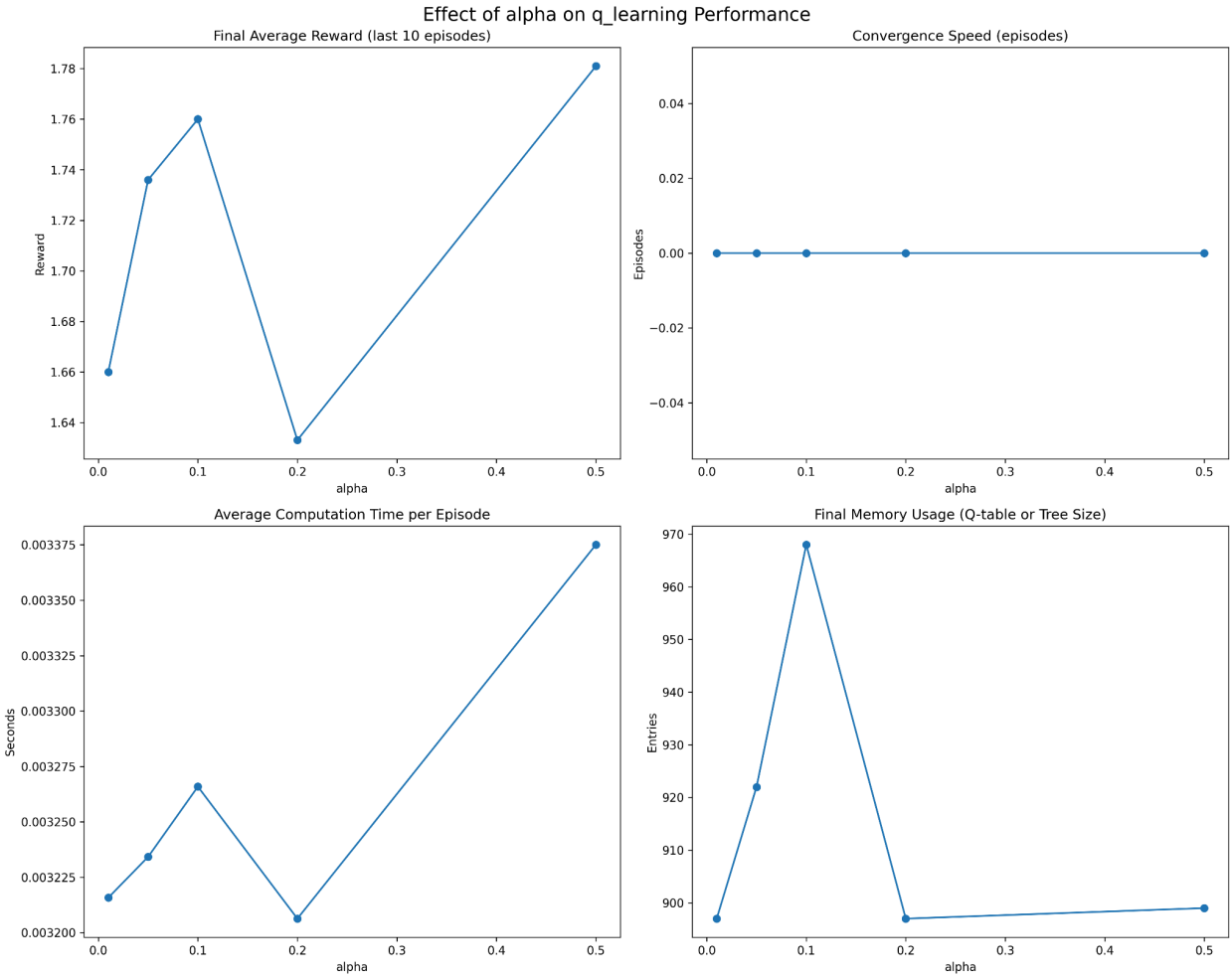

Figure 5: Impact of Learning Rate (α) on Q-Learning Performance

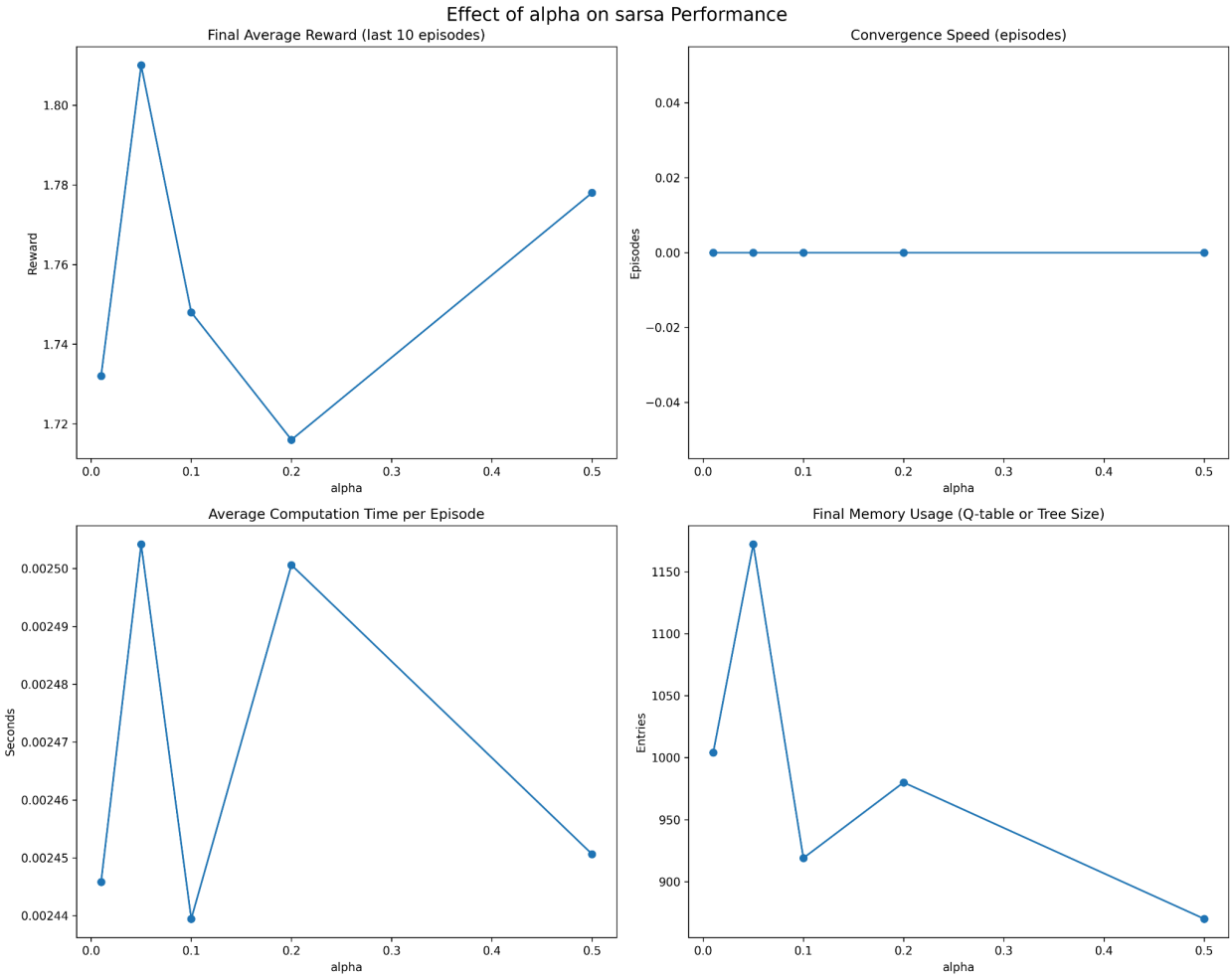

Figure 6: Impact of Learning Rate (α) on SARSA Performance

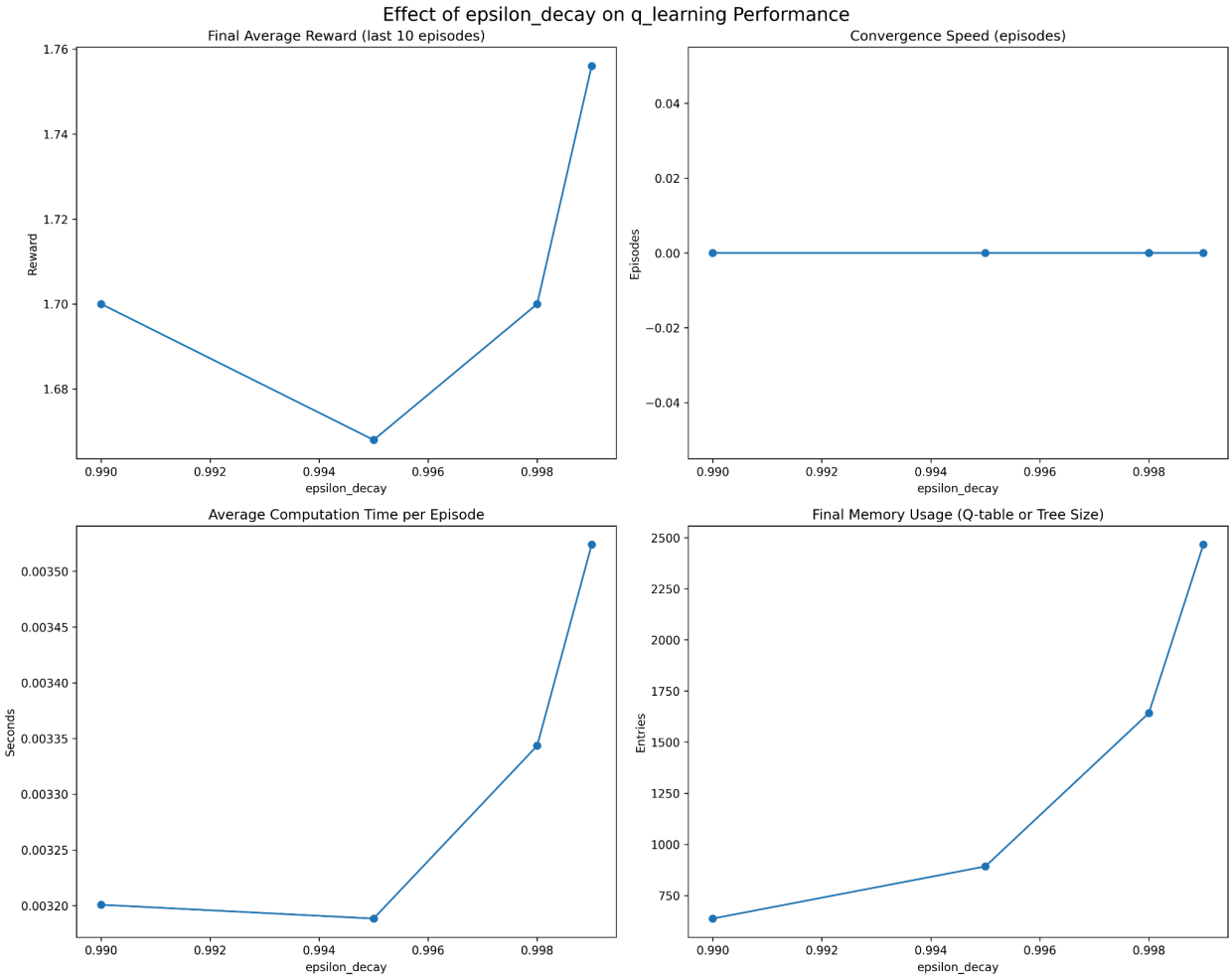

Figure 7: Impact of Epsilon Decay (ε) on Q-Learning Performance

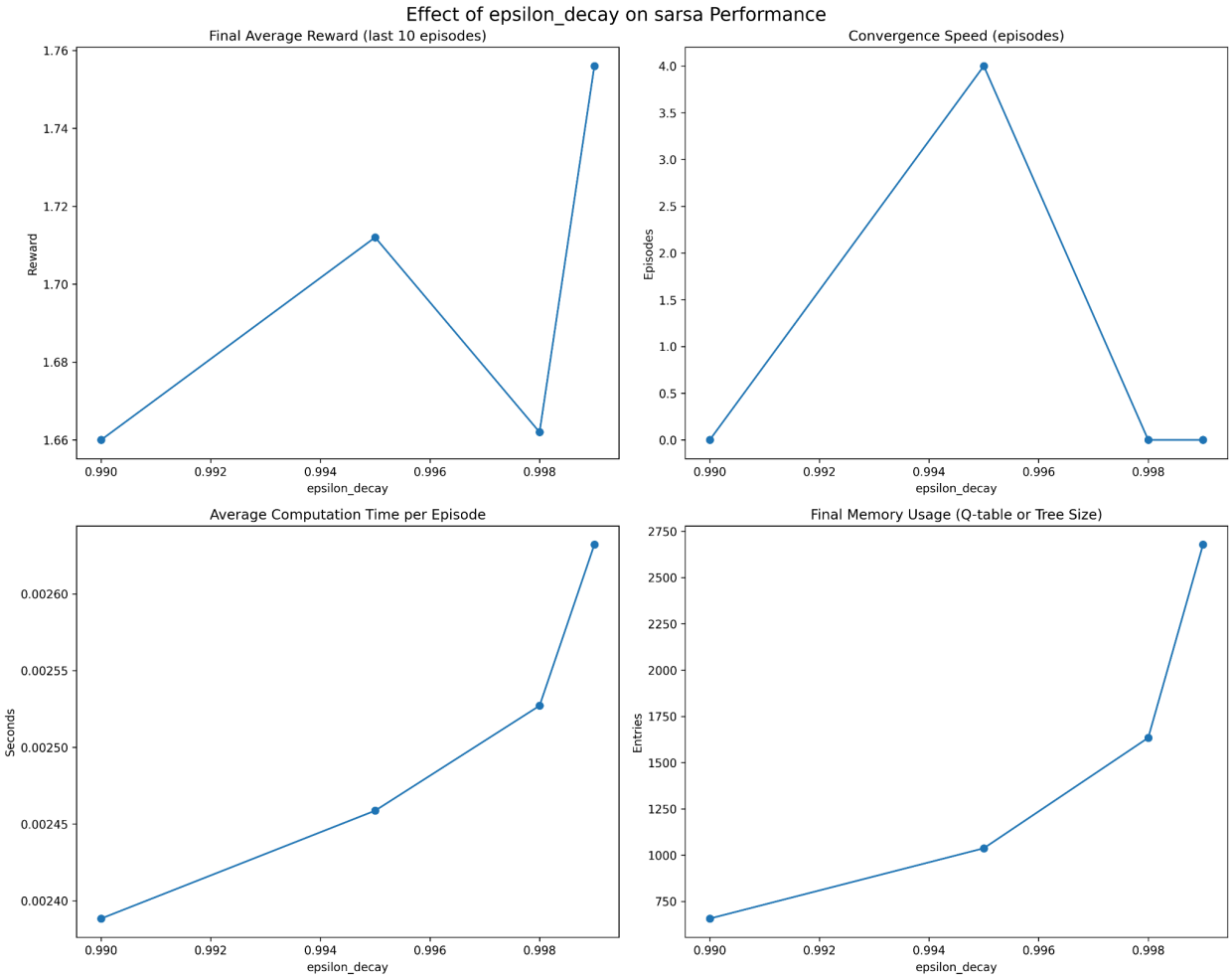

Figure 8: Impact of Epsilon Decay (ε) on SARSA Performance

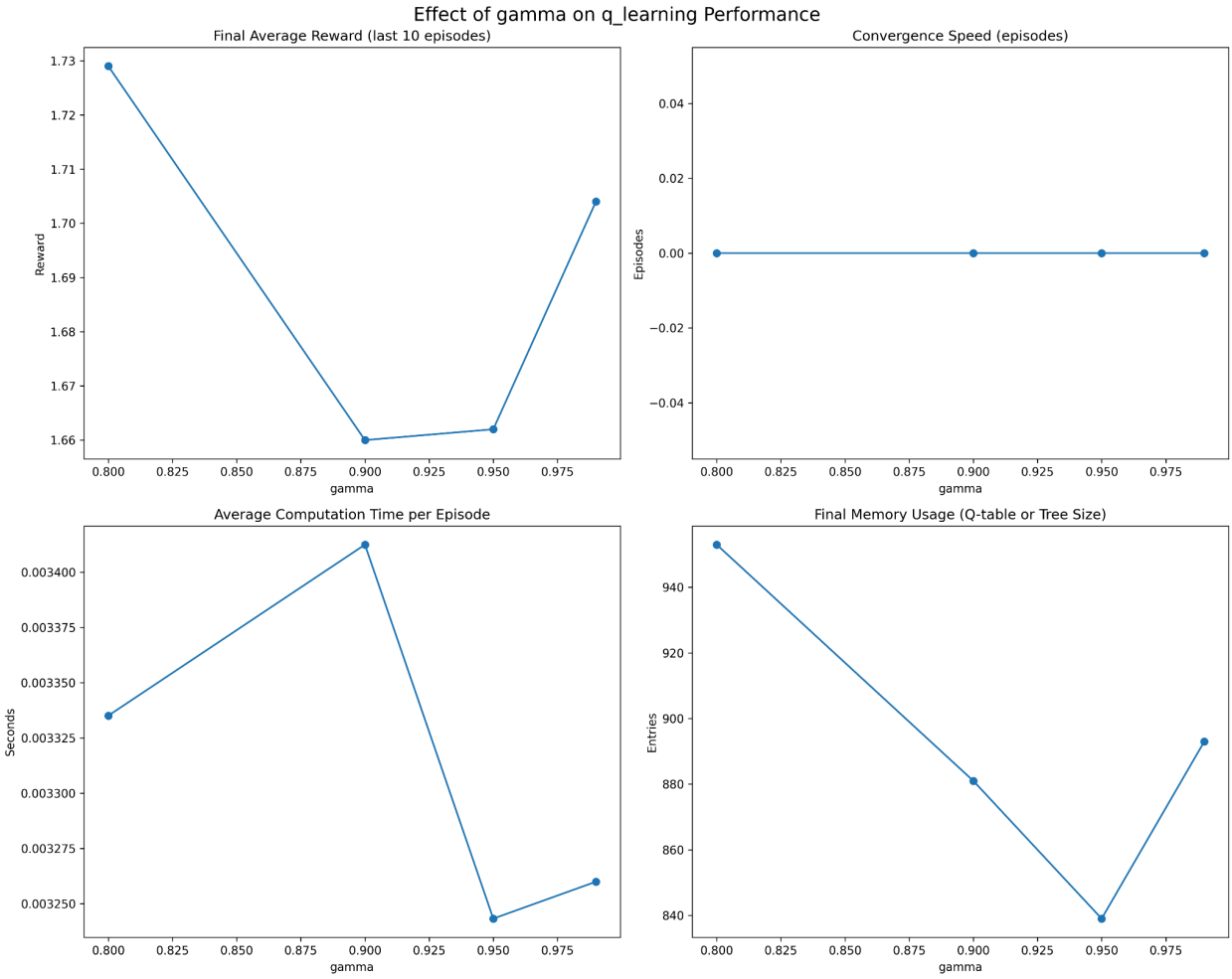

Figure 9: Impact of Discount Factor (γ) on Q-Learning Performance

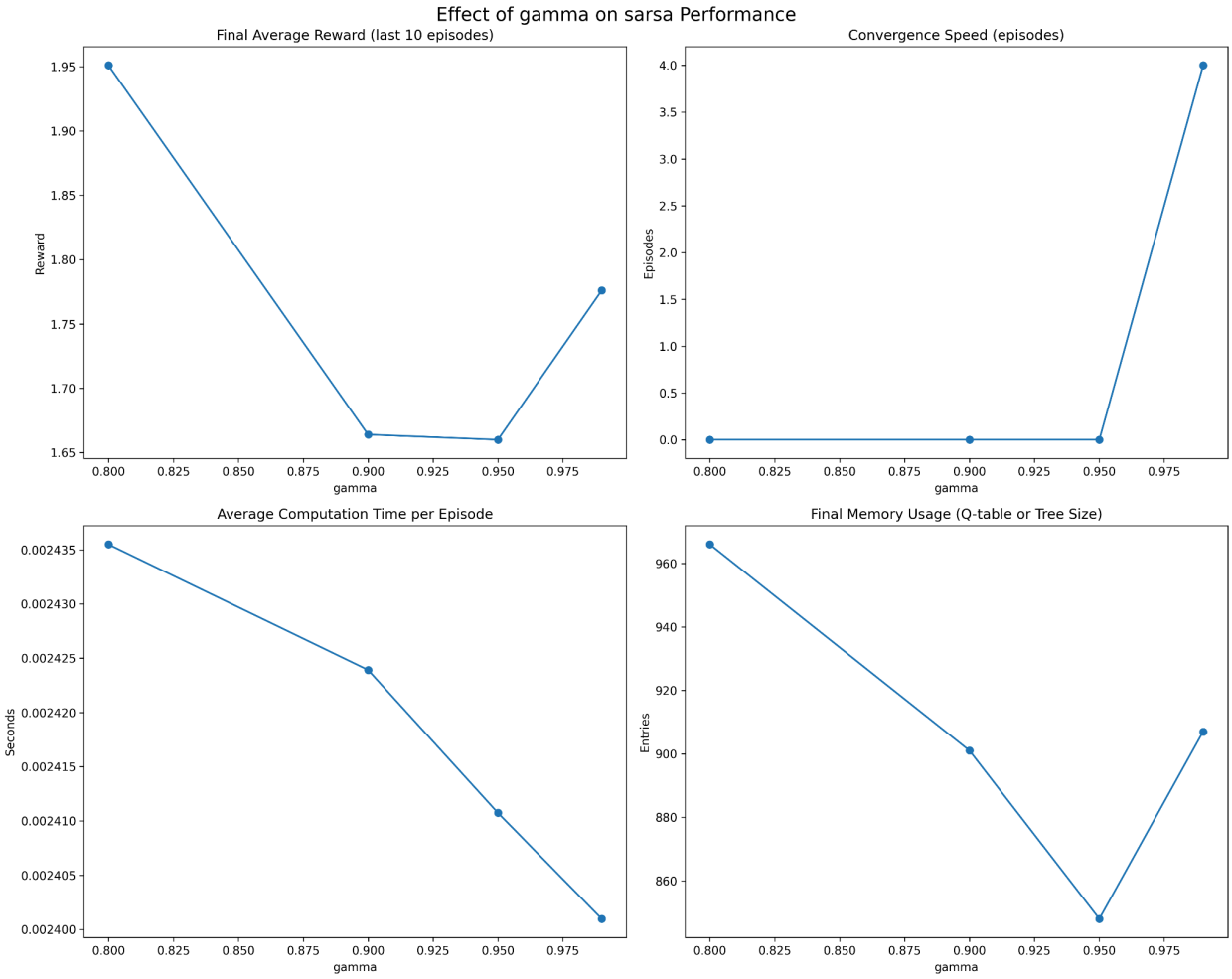

Figure 10: Impact of Discount Factor (γ) on SARSA Performance

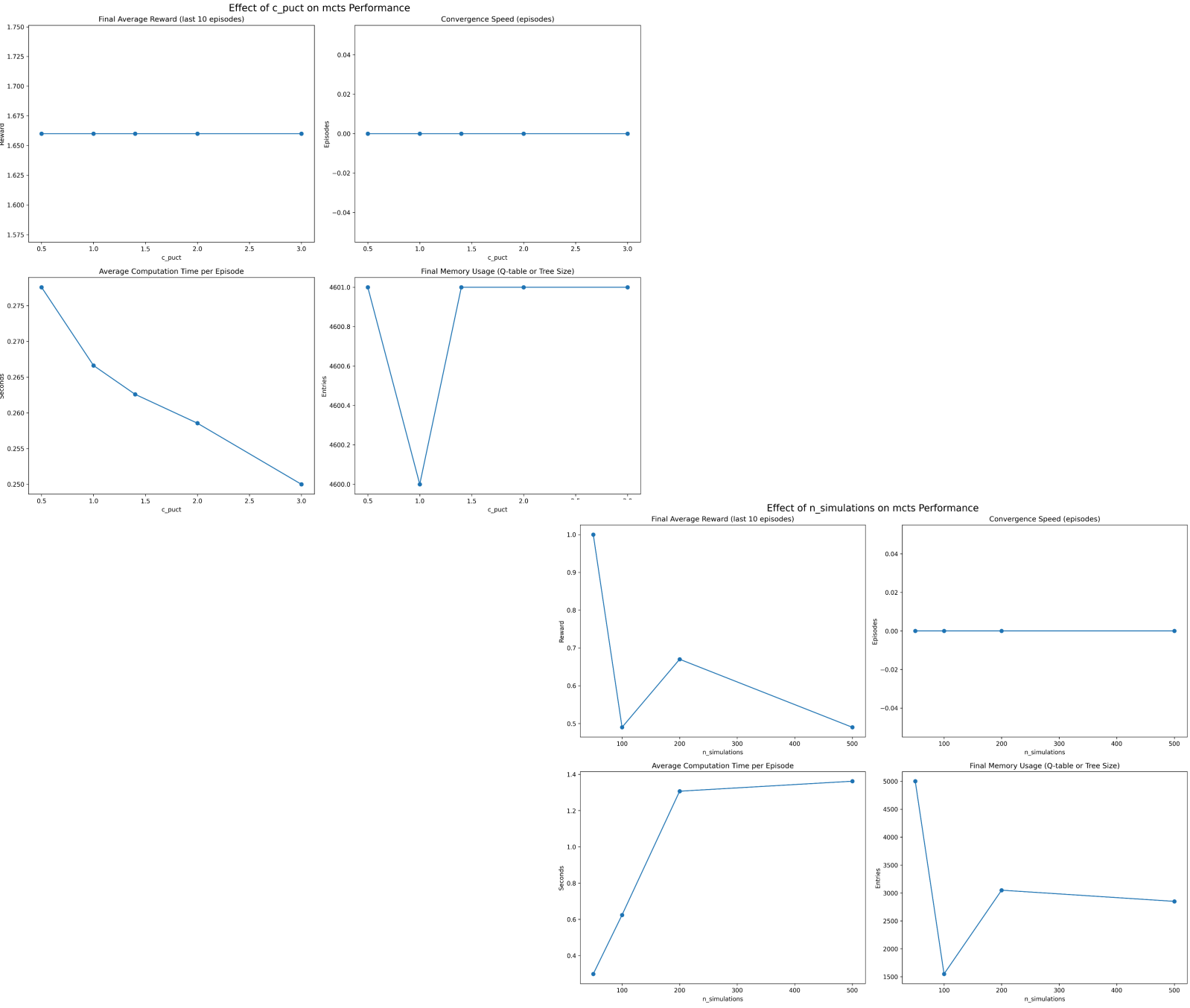

Figure 11: MCTS Performance vs. Number of Simulations

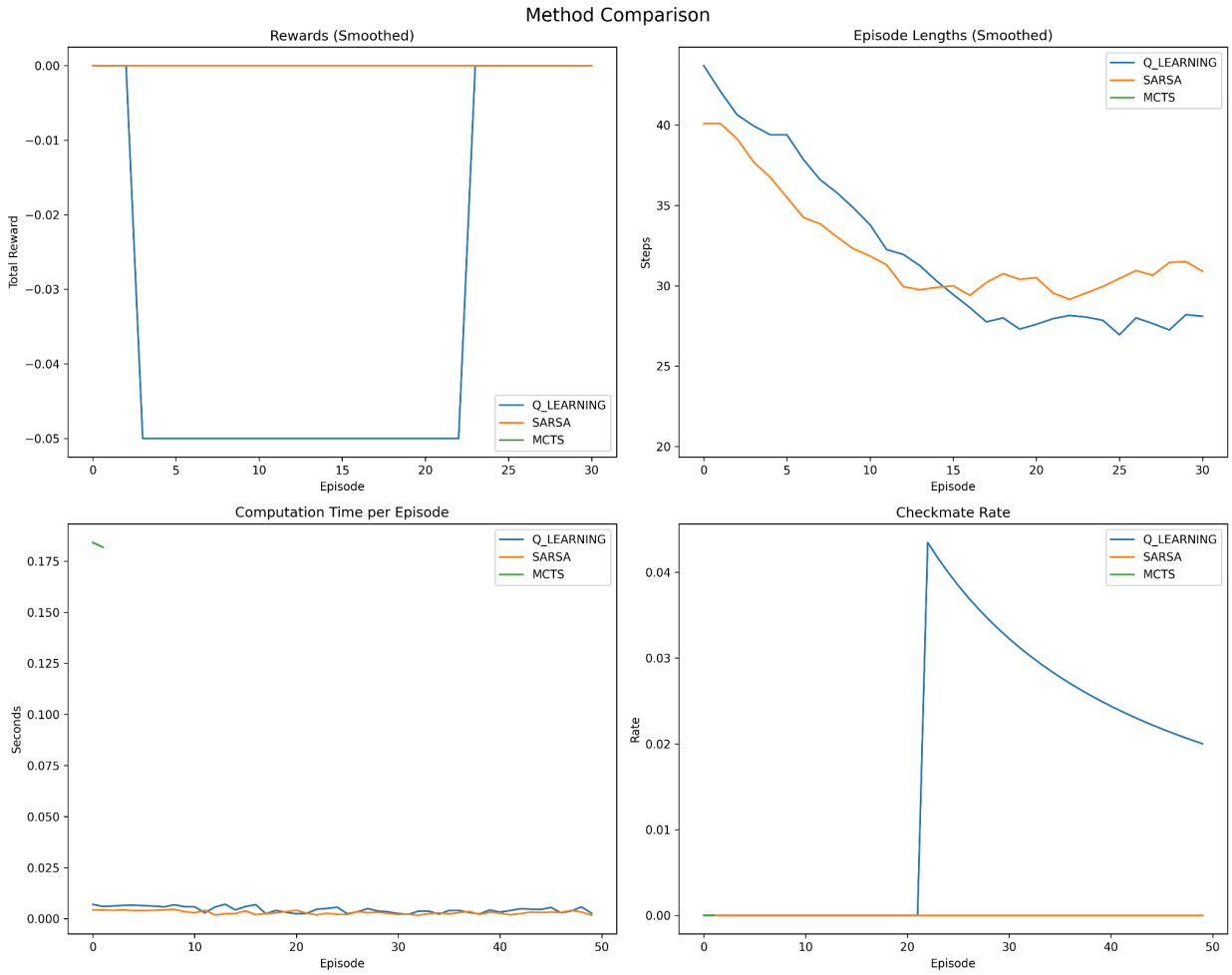

Figure 12: Final Performance of MCTS, Q-Learning, and SARSA After Tuning