Playing Pong Using Multiple RL Alogrithms

Kate Hornbeck and Alayna Stepp

Abstract This research project investigates the application of multiple reinforcement learning (RL) algorithms - Q-learning, SARSA, and Monte Carlo - in training agents to play the classic Atari game of Pong. The game environment was simplified into a discrete grid-based system to simplify the state and action spaces, enabling agents to learn and optimize their strategies through trial and error. Agents were trained against a consistent, deterministic opponent (a wall that always returns the ball), providing a stable feedback loop to facilitate effective learning. For further analysis, the trained agents were introduced to a dynamic, variable environment featuring agent-versus-agent competition, testing their adaptability to less predictable scenarios. Experimental results revealed that the Q-learning agent outperformed SARSA and Monte Carlo algorithms in both static and dynamic environments, demonstrating superior adaptability and strategy optimization. This study contributes valuable insights into the comparative performance of RL algorithms in discrete action spaces and controlled game environments. GitHub at: https://github.com/alaynastepp/jhu-reinforcement-learning.

1 Introduction

As a core area of artificial intelligence (AI), reinforcement learning (RL) enables algorithms to find patterns and make decisions that optimize the desired end goal, usually framed in winning a game. This paper takes a deeper dive into different RL algorithms and their training behaviors on the classic Atari game, Pong. The selection of an appropriate RL algorithm is highly dependent on the specific application, therefore the goal of this analysis is to study three different RL algorithms' adaptability and performance in this simplified game environment. By analyzing the training behaviors of RL algorithms in Pong, this study looks to unveil patterns that can be applied to similarly restricted 2-dimensional paddle-and-ball games and contribute to the growing body of knowledge in AI research.

Creating a static environment for training with the application of the policies to a dynamic environment allows further analysis in algorithm exploration and generalization. Increasing the diversity of experience in a static environment permits a policy to grow and become applicable to a variety of unknown states. An agent aware of the optimal action for a few amount of states exploits the known information to

increase accuracy in a small area of state space, but struggles with generalizing. The ideal algorithm finds the balance between exploration and exploitation to optimize long-term performance.

The distinctive capability to explore and optimize independently of the current policy is believed to allow Q-learning to converge faster and find more robust strategies. The SARSA algorithm contributes more of a balance of exploration and exploitation due to its on-policy approach. It is hypothesized that this will allow the algorithm to perform moderately well in this environment. Lastly, Monte Carlo is expected to face challenges in dynamic environments, given its reliance on episodic learning and complete trajectory updates.

2 Background

Reinforcement learning is a branch of machine learning in which agents learn optimal strategies through interaction with an environment guided by rewards and penalties. Unlike supervised learning, RL emphasizes exploration and sequential decision-making, making it particularly suited for tasks involving dynamic environments and long-term planning. Algorithms such as Q-learning, SARSA, and Monte Carlo are foundational to RL, each employing unique mechanisms for updating knowledge and improving policies.

Games have historically served as a valuable platform for RL research due to their well-defined rules, measurable outcomes, and structured environments. Pong, one of the earliest video games, stands out as an ideal candidate for RL experimentation. Its simplicity, coupled with its discrete state and action spaces, allows researchers to isolate algorithmic performance without excessive computational overhead.

Despite significant advancements in RL research, questions remain regarding the relative performance of different algorithms in dynamic, multi-agent scenarios. With this in mind, our research project attempts to answer the question: How do Q-learning, SARSA, and Monte Carlo algorithms compare in their ability to train agents for playing Pong in both static and dynamic environments?

2.1 Agents

To ensure consistency across all reinforcement learning algorithms, we began by developing a Python file that defined a generic agent structure. This file established common functions, such as action selection, reward accumulation, and policy updating, which were shared across all agents. These functions were invoked in the main script during evaluation, enabling seamless integration of different agent types.

We then implemented three specific agents - SARSA, Monte Carlo, and Q-learning - each tailored to a specific reinforcement learning algorithm.

Monte Carlo The Monte Carlo agent uses episodic learning, updating its action-value estimates at the end of each episode based on the total accumulated reward. This method is well-suited for environments where episodes naturally terminate. By averaging returns over multiple episodes, Monte Carlo methods converge to the optimal policy in a model-free manner[1]. However, this episodic dependency can make it less effective in dynamic or non-terminating environments.

SARSA The SARSA agent utilizes an on-policy temporal difference control method, where the agent learns action-value pairs ($Q(s, a)$) by interacting with the environment and following its current policy. By updating its Q-values based on the sequence of states, actions, rewards, and the next action (s, a, r, s', a'), SARSA emphasizes balancing exploration and exploitation during training[2]. This approach makes it particularly effective in environments with stochastic state transitions.

Q-learning The Q-learning agent employs an off-policy temporal difference learning approach, allowing it to learn the optimal policy independently of the actions taken. By updating Q-values using the formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)],$$

the agent focuses on maximizing the expected future reward[3]. This off-policy nature enables greater exploration and robustness, making Q-learning a powerful algorithm in reinforcement learning.

2.2 Atari Pong

Pong, originally released in 1972 by Atari[4], is a two-dimensional video game inspired by table tennis. Created as a training project for programmer Al Alcorn under the guidance of Atari founder Nolan Bushnell, Pong became a groundbreaking success that established video games as a profitable industry. The gameplay involves two paddles and a bouncing ball, with players moving their paddles vertically to return the ball and prevent it from passing their paddle. The ball's return angle and speed can be controlled by hitting it at different parts of the paddle and movement of the paddle on impact. If a player manages to get the ball past their opponent, they score a point. The simplicity of its mechanics, combined with competitive gameplay and strategic elements, has made Pong a significant milestone in video game history.

1. Sutton and Barto 2018.
2. Sutton and Barto 2018.
3. Watkins 1989.
4. Modany 2012.

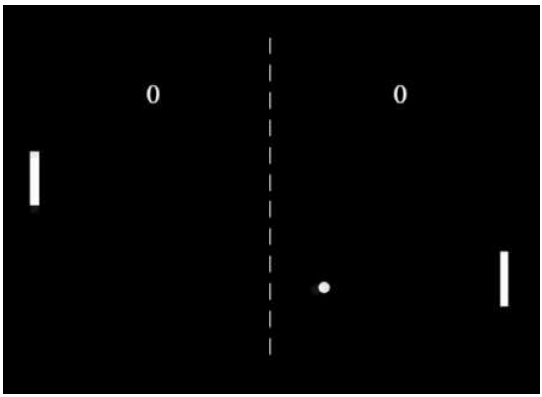


FIGURE 1. *Original Atari Pong Graphic*

3 Prior Literature

The use of reinforcement learning in games has been extensively explored in previous research, with notable successes such as AlphaGo and Deep Q-Networks (DQN) mastering Atari games[5]. Simplified games like Pong have been widely used in RL studies due to their structured environments and clear evaluation metrics. The game has served as a testbed for RL algorithms due to its simplicity, making it ideal for understanding RL performance in controlled environments.

Previous work has shown that RL algorithms, such as Q-learning and SARSA, often perform well in discrete state spaces, as seen in *OpenAI Gym*'s use of Pong as a benchmark[6]. These algorithms, which rely on different exploration-exploitation strategies, have been tested in Pong environments and shown to learn optimal strategies for ball interception and return. In addition, Monte Carlo methods have also been explored in Pong, particularly in domains requiring episodic learning, though they have been found to be less effective in real-time decision-making tasks compared to Q-learning and SARSA[7].

However, limited research directly compares these algorithms in a grid-based abstraction of Pong, especially in environments that vary in complexity. For example, while Q-learning is known for its off-policy nature, which allows for independent exploration and optimization[8], it remains unclear how such methods perform when transitioning between static and dynamic environments. In our study, we address this gap by evaluating the performance of Q-learning, SARSA, and Monte Carlo methods

5. Mnih et al. 2015.
6. Brockman et al. 2016.
7. Sutton and Barto 2018.
8. Sutton and Barto 2018.

in both static and dynamic versions of a grid-based Pong environment. By training agents in these two distinct settings, we contribute valuable insights into how these algorithms adapt to different levels of unpredictability.

4 Methods

4.1 Environment

To evaluate the performance of different RL algorithms, we developed two distinct training environments:

1. **Static Opponent Environment:** The agent competes against a wall that consistently returns the ball, creating a predictable training setup with minimal variability.
2. **Dynamic Opponent Environment:** The agent competes against another agent, introducing unpredictability and requiring the RL algorithms to adapt to dynamic strategies.

In our study, we trained and evaluated agents using the *Static Opponent Environment*, where the agent competes against a "perfect" opponent that always returns the ball, creating a controlled and predictable environment. This environment enforces inexperienced agents to take responsibility for their own actions, ensuring they gain experience independently rather than inadvertently benefiting due to the poor performance of another agent.

We added the *Dynamic Opponent Environment* to examine how pre-trained agents, trained in the static environment, would perform when faced with the added complexity of another agent whose actions are not fixed. The dynamic environment introduces variability in the agent's behavior, which could challenge the strategies learned in the static environment.

It is important to note that using the *Dynamic Opponent Environment* for actual evaluation is not appropriate for assessing each agent's performance. Since the agents were pre-trained in the static environment, they may have overfitted to the specific, predictable nature of that setup. In the dynamic environment, the agents are essentially tested on their ability to adapt to an environment that they were not trained for, which introduces several factors that could bias the evaluation:

- **Transferability of learned strategies:** The strategies learned in the static environment may not fully generalize to the dynamic environment, where unpredictability plays a key role. For example, a strategy that works well when the opponent always behaves the same way may not be effective when the opponent introduces randomness or new strategies.
- **Overfitting to a simple model:** The agent may have overfitted to the "perfect" opponent in the static environment, learning a narrow strategy that works in that specific setup but struggles when faced with a more dynamic, human-like opponent. For example, the static environment returns the ball to the agent maintaining its

previous angle of velocity, while a dynamic agent can change the direction based on the location of impact on the paddle.

- **Unrealistic performance expectations:** The pre-trained agents were not optimized for the dynamic environment, meaning their performance in this setting could be misleading when compared to agents trained directly in the dynamic environment. As a result, the agents' true capabilities in adapting to dynamic environments may be underrepresented.

Thus, while the dynamic environment provides valuable insight into how pre-trained agents respond to a new challenge, it should not be used as the sole method of evaluation for each agent's performance. This could lead to an inaccurate assessment of their overall capabilities in handling more complex environments.

The state space representation is an integer containing the following information: x-axis ball position, y-axis ball position, y-axis paddle position, and ball x and y velocity directions. To restrict the state space, the game existed on a 10x10 grid where the first and last columns contained the agent paddle or the static wall agent. This makes up a total 9000 possible states, 810 of which are terminal, for a single agent operating on one side of the environment. The starting state of the environment is randomized such that the ball y position and x and y velocity directions are changing to increase variability.

4.2 Interface: Parameters and Configuration

To allow for flexible experimentation, we implemented a series of configurable parameters using the Python `argparse` library. These parameters provided options for choosing algorithms, visualizing results, setting hyperparameters, and saving agents. Table 1 summarizes the parameters and their functionality.

Parameter	Description
`-sarsa`	Runs the SARSA algorithm.
`-monte`	Runs the Monte Carlo algorithm.
`-qlearning`	Runs the Q-learning algorithm.
`-viz`	Enables a visualization of the gameplay.
`-plot`	Generates plots of rewards over episodes.
`-gamma`	Sets the discount factor (γ).
`-alpha`	Sets the learning rate (α).
`-epsilon`	Sets the exploration rate (ϵ).
`-left`	Places the agent on the left side of the board.
`-right`	Places the agent on the right side of the board (default).
`-save`	Saves the agent's q table after training for reuse as a pre-trained agent.

TABLE 1. *Configurable parameters used in the implementation.*

4.3 Agents

Agent Position

The `-left` and `-right` parameters determined the side the agent will be on. This flexibility allows testing agents in varying setups, particularly in the agent vs. agent environment, where symmetry played a critical role in fair evaluation.

Agent Training

For each RL algorithm mentioned above, Monte Carlo, SARSA, and Q-Learning, both an agent acting on the left paddle and one on the right paddle was trained. Each single agent was trained against an opposing wall, or static environment for 1000 episodes, each episode/game lasting at maximum 1000 steps, or time units in this simulation. If an agent reached the maximum amount of steps and did not allow the ball to pass its paddle, the episode was concluded as a victory. If an agent allowed for the ball to pass it's paddle, the episode was concluded as a loss.

Pre-Trained Agents

The `-save` parameter enabled the saving of trained agent q tables for reuse. This feature was crucial for testing how pre-trained agents performed in different environments, particularly the agent vs. agent setting. After loading a pre-trained q table, the agent was still permitted to adjust policy values.

Hyperparameters

The hyperparameters chosen allowed for precise control over the learning process, facilitating experimentation with different algorithmic behaviors. Three key hyperparameters were included:

- `-gamma` (γ): The discount factor, determining the weight of future rewards relative to immediate rewards.
- `-alpha` (α): The learning rate, controlling how much the agent updates its knowledge based on new experiences.
- `-epsilon` (ϵ): The exploration rate, dictating the likelihood of the agent exploring new actions versus exploiting known strategies.

Hyperparameter Tuning

To identify the best values for gamma, epsilon, and alpha in each type of agent, the hyperparameter tuning approach was taken. Utilizing a grid search methodology, each agent iterated through a different set of potential parameters and ran for 1000 episodes each. Each step in the grid search took the approach described in the Agent Training section above.

4.4 Visualization (`-viz`)

The visualization feature, implemented using `pygame`, provides a graphical representation of the gameplay. Figure 2 shows examples of the visualizations for the two environments.

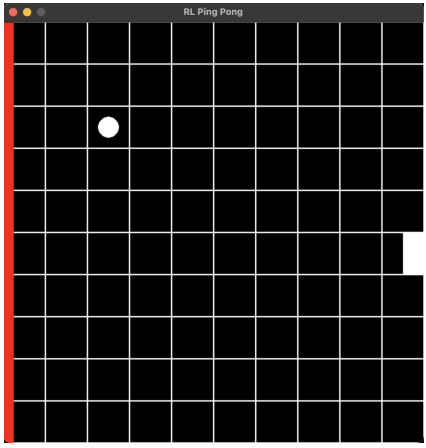
5 Results

5.1 Hyperparameter Tuning

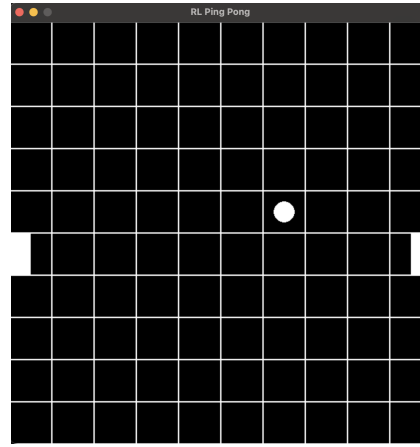
Before conducting the main experiments, we performed hyperparameter tuning to identify the optimal parameters for each agent. All agents were tuned while playing on the right side of the board, as initial results showed similar performance for agents regardless of side. The optimal parameters found for each agent are summarized in Table 2.

These optimal hyperparameters give us some insights into the benefits of each algorithm. The low exploration rate, epsilon, value for every algorithm suggests a quick exploitation of learned policies. Q-Learning benefits from a very high discount factor, gamma, which confirms its capability to focus on long-term rewards and higher reliance on exploration. These parameters were used in all subsequent experiments and analyses, ensuring a fair comparison between the agents under their best-performing configurations.

After finding the optimal parameters for each agent, we ran each agent again,



Static Opponent Environment - Perfect opponent environment (wall) with the agent on the right.



Dynamic Opponent Environment - Agent vs. agent environment with paddles on both sides.

FIGURE 2. *Visualization of the gameplay using* `pygame`.

Agent	Alpha	Gamma	Epsilon	Avg Reward
Monte Carlo	0.8000	0.9000	0.0100	190.22
SARSA	0.1000	0.5000	0.0100	194.64
Q-Learning	0.8000	0.9500	0.0100	43.50

TABLE 2. *Hyperparameter Tuning results for each algorithm: The values for alpha, gamma, and epsilon correspond to the learning rate, discount factor, and exploration rate, respectively. Average reward is the reward per episode achieved by the agent after converging with the tuned hyperparameters.*

each with their respective best parameters, and saved their Q-tables off to use as pre-trained agents in the agent vs agent environment. Loading pre-trained agents into that environment included loading in what parameter values they were trained with, in order to keep their performance consistent and optimal.

5.2 Performance by Winning Percentage

Figure 3 shows the winning percentage of the three agents—Monte Carlo, SARSA, and Q-Learning—over 1000 episodes. Winning percentage is calculated as

$$\text{Winning Percentage} = \frac{\text{Total Wins}}{\text{Agent Count} \times \text{Episode Count}}$$

The trends in the graph reveal key differences in the learning behaviors of each algorithm.

Monte Carlo learning shows a rapid increase in winning percentage during the first

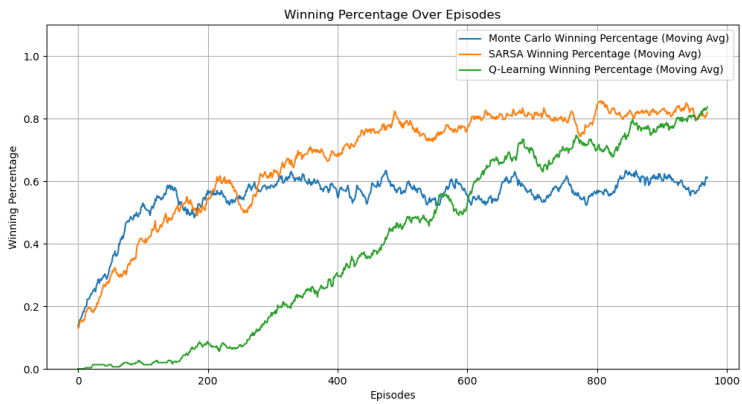


FIGURE 3. *Performance of Monte Carlo, SARSA, and Q-Learning agents over 1000 episodes, measured by winning percentage.*

200 episodes, starting at around 0.1 and rising to approximately 0.6. However, after this initial period, the winning percentage plateaus, oscillating slightly but remaining around 0.6 for the remainder of the training. This behavior can be attributed to the algorithm's reliance on episodic updates. It learns effective policies for specific trajectories early on, allowing it to quickly adapt to straightforward patterns in the environment. However, its inability to incrementally refine policies during episodes means it struggles with more dynamic or diverse scenarios, leading to stagnation in performance beyond its initial improvement.

SARSA demonstrates a more consistent growth, starting at 0.2 and reaching approximately 0.8 within the first 500 episodes. After this point, its performance stabilizes near 0.8, with minimal oscillations. SARSA's success can be attributed to its on-policy nature, which enables it to learn directly from the policies it executes. This approach allows it to develop robust strategies that generalize better to dynamic environments compared to Monte Carlo. The plateau at 0.8 suggests that SARSA converges to an effective, albeit not optimal, policy for the game.

Q-Learning starts with a very low winning percentage (close to 0) but exhibits steady growth throughout the training period. By the end of 1000 episodes, its winning percentage approaches 0.8, comparable to SARSA, but with a steeper growth curve and more oscillations. The slow start can be explained by Q-Learning's off-policy approach, which requires more exploration to learn optimal actions. Its steady growth reflects its ability to incrementally learn the value of actions in states it explores, even when these states are not directly visited. Q-Learning adapted better than Monte Carlo and SARSA to a randomized environment, but its slower learning process meant it took longer to reach comparable performance levels.

5.3 Performance by State Visitation

Figure 4 shows the cumulative percentage of states visited overtime of the three agents-Monte Carlo, SARSA, and Q Learning-over 1000 episodes. Similarly to the winning percentage, we can see trends in the graph that are key to the differences in the algorithms learning behaviors.

Monte Carlo and SARSA have very low visit percentages throughout the 1000 episodes. They both appear to asymptote around 5% of visited states, meaning exploration is very low for these algorithms. In contrast, Q-learning has a linear increase for approximately the first 600 episodes then decreases exploration rate. If these trends continue for more episodes, Monte Carlo and SARSA would not explore any more state space but Q-learning does not appear to asymptote therefore could continue to steadily increase.

5.4 Performance in Agent vs Agent Environment

After tuning the hyperparameters and training each agent with their optimal parameters, we evaluated their performance against each other in an agent vs. agent environment. The pre-trained agents were loaded with the parameter configurations that achieved

FIGURE 4. *Performance of Monte Carlo, SARSA, and Q-Learning agents over 1000 episodes, measured by state visit percentage.*

the best performance in the agent vs. wall scenarios, ensuring consistent and optimal conditions during these evaluations. Below, we present the experimental metrics for the agent vs. agent matches.

TABLE 3. *Agent vs Agent Win Rates*

Match Combination	**Agent 1**	**Agent 2**	Agent 1 Win Rate (%)	Agent 2 Win Rate (%)
SARSA vs Monte Carlo	SARSA	Monte Carlo	21.20	20.70
SARSA vs Q-Learning	Q-Learning	SARSA	43.60	39.00
Q-Learning vs Monte Carlo	Q-Learning	Monte Carlo	46.00	23.00

In the results presented in Table 3, it is notable that the win rates for the two agents in each match combination do not sum to 50%. This observation arises from the specific definition of a "win" in our environment. A win is defined as a situation where the game concludes with an agent achieving a positive reward. Consequently, if neither agent achieves a positive reward by the end of an episode, the outcome is recorded as a "loss" for both agents.

This dynamic reflects the competitive and resource-constrained nature of the environment. Unlike many traditional zero-sum games where one player's gain is always the other's loss, our setup allows for scenarios where both agents can fail to

achieve their objectives. This unique characteristic of the environment can be seen in the individual match outcomes, which are analyzed below:

1. **SARSA vs Monte Carlo**: In this match, SARSA demonstrated a slight edge over Monte Carlo, achieving a higher average reward and win rate. This may be attributed to SARSA's policy updates, which balance exploration and exploitation in a way that adapts more effectively to an opponent's actions. The low state-action pair visitations suggest limited exploration, implying that both agents primarily relied on familiar strategies rather than actively exploring the entire state space.
2. **SARSA vs Q-Learning**: Q-Learning outperformed SARSA in this scenario, suggesting that its strategy for learning state-action values with a higher learning rate and an emphasis on temporal difference learning was more effective in this competitive environment. The higher reward and win rate for Q-Learning align with its ability to make more informed decisions and adjust more efficiently compared to SARSA.
3. **Q-Learning vs Monte Carlo**: Q-Learning again demonstrated superior performance, which aligns with its improved learning mechanism. While Monte Carlo can achieve higher rewards during training in static conditions, it struggles with the dynamic nature of the agent vs. agent environment. The percentage of state-action pairs visited indicates that Monte Carlo explores a broader range of states compared to the others but lacks the strategic updates necessary to secure wins.

Overall, these results show that agents that could adapt to changing conditions and learned from interactions were more successful. Q-Learning's ability to incrementally update its knowledge with a higher learning rate and its adaptability under different states allowed it to consistently outperform the other agents. The consistency of results in the agent vs. agent environment highlights the importance of effective learning strategies and parameter tuning in achieving competitive performance.

6 Discussion

6.1 Agent Performance and Randomized Starts

Initially, Q-learning appeared to outperform both SARSA and Monte Carlo in our Pong environment. This early observation stemmed from our use of a deterministic starting condition for each episode, where the ball always began at the exact center of the grid and moved in the same direction. Under these controlled conditions, Q-learning demonstrated rapid convergence and consistent high performance, reinforcing its reputation as a robust and efficient reinforcement learning algorithm[9].

However, when we implemented a randomized starting condition for each

9. Sutton and Barto 2018.

episode—where the ball's y position was chosen randomly within the grid (excluding the very top), and its direction was set randomly—the performance rankings shifted dramatically. While the ball's x position was still initialized at the center of the grid to ensure fairness and avoid bias from immediate proximity to either agent, Q-learning's performance dropped significantly, whereas Sarsa and Monte Carlo agents exhibited noticeable improvements.

This performance discrepancy suggests that Q-learning may rely heavily on exploiting deterministic patterns present in the environment. In the absence of variability, Q-learning's off-policy nature and ability to maximize over the action space allow it to quickly identify and exploit repetitive conditions. However, when presented with increased randomness, Q-learning struggled to generalize its learned policy to adapt to the broader state-action space effectively. This behavior aligns with findings from prior research, indicating that Q-learning may overfit to specific environmental structures in controlled setups[10].

In contrast, SARSA's on-policy approach, which updates the policy based on the actions it actually takes, appears better suited for environments with inherent randomness. Similarly, Monte Carlo's episodic updates, which average returns over multiple trajectories, allow it to develop more generalized strategies that are less dependent on specific starting conditions. These attributes likely contributed to their improved performance in the randomized setup.

This shift in agent performance highlights the importance of designing training environments that accurately reflect the diversity and unpredictability of real-world conditions. Over-reliance on deterministic setups risks overestimating the efficacy of algorithms like Q-learning, which may excel under controlled circumstances but falter when faced with increased variability. Future studies should prioritize randomized or dynamic conditions to ensure that agents are evaluated based on their ability to adapt and generalize, rather than simply exploit fixed patterns.

6.2 Agent Strategies and Observed Behaviors

Beyond the statistical performance differences, visualizing the behaviors of each agent revealed notable distinctions in their strategies. Both SARSA and Monte Carlo agents tended to adopt a stationary strategy: they would position themselves in a fixed spot and wait for the ball to reach a predictable angle. This behavior often resulted in the ball bouncing back to the same square repeatedly, effectively creating a stable loop. In contrast, the Q-learning agent displayed a more dynamic strategy, actively moving around the grid to intercept the ball, even in scenarios where it wasn't strictly necessary. We can see these results in Figures 5 and 6.

This raises an interesting question: Is the stationary behavior of SARSA and Monte Carlo agents a form of "cheating" or simply an intelligent exploitation of the

10. Watkins 1989.

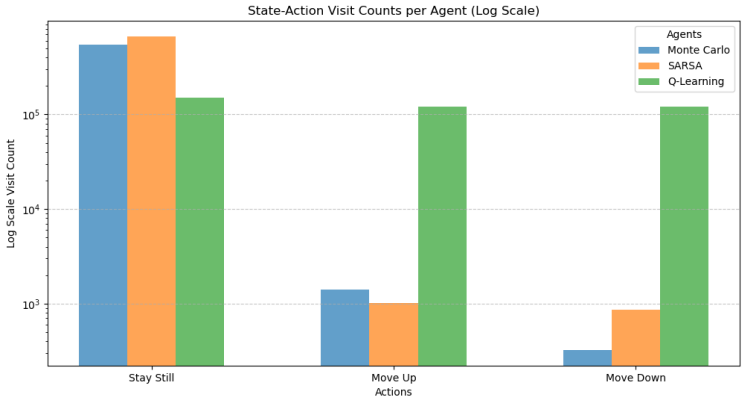


FIGURE 5. *Analysis of state-action visits of Monte Carlo, SARSA, and Q-learning agents over 1000 episodes.*

game mechanics? From one perspective, this stationary strategy minimizes energy expenditure (or, in this case, unnecessary movement) while still achieving the goal of returning the ball. By leveraging predictable patterns in the ball's trajectory, these agents effectively "game the game" to maximize efficiency. This aligns with the principle of exploiting known environmental dynamics for optimal outcomes[11].

On the other hand, this behavior might be considered a form of overfitting to the specific environment and rules of the game. The agents do not exhibit adaptive behaviors that could generalize to variations in gameplay, such as an opponent introducing erratic movements or changes in the ball's trajectory. In real-world scenarios, where environments are rarely as controlled or predictable, such strategies may prove inadequate.

Q-learning's more dynamic approach, while less efficient under deterministic conditions, suggests a greater degree of exploration and adaptability. By actively moving to different positions, the Q-learning agent appears to prioritize learning generalizable policies over simply exploiting current conditions. However, as discussed earlier, this tendency to explore comes at the cost of performance when the environment is deterministic, as it struggles to leverage consistent patterns as effectively as SARSA and Monte Carlo.

This divergence in strategies underscores the trade-offs inherent in reinforcement learning algorithms. While SARSA and Monte Carlo excel in exploiting stable environments, their lack of exploration may limit their robustness in more complex or unpredictable scenarios. Conversely, Q-learning's exploratory nature may allow it

11. Sutton and Barto 2018.

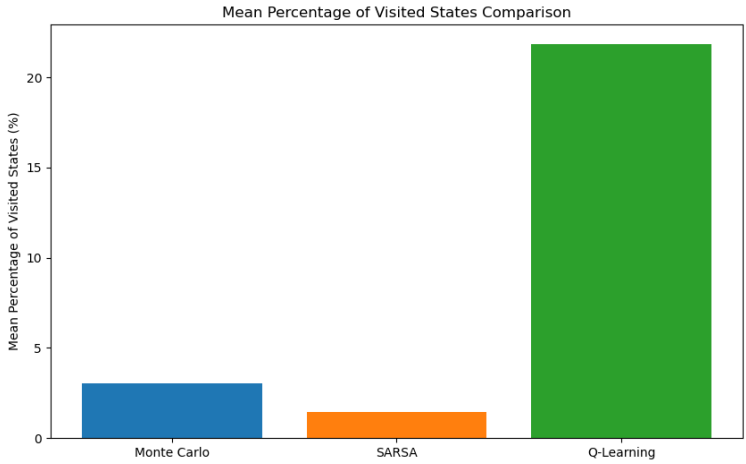


FIGURE 6. *Analysis of state space exploration of Monte Carlo, SARSA, and Q-learning agents over 1000 episodes.*

to adapt better to variability, though this comes with the risk of reduced efficiency in controlled setups. Understanding and balancing these trade-offs is crucial for designing RL agents capable of handling diverse real-world challenges.

7 Future Research

While this study provides valuable insights into SARSA, Q-Learning, and Monte Carlo algorithm application in a dynamic environment utilizing a static environment for training, several experiments and modifications would allow for further research. While the visualization tool helped exponentially in deciphering behaviors of the algorithms, analysis should be focused on specific behavior identification. For example, does the agent wait until the last minute to move the paddle or consistently optimizing their chances of hitting the ball. This leads to another future research area of algorithm "comfort zones." The work done in this paper specifically points out the large difference in state exploration between the algorithms, analysis needs to be done to decipher if this is a benefit or a fault of the algorithm.

In regards to the simple environment implemented, further research could explore the use of reward shaping and curriculum learning to improve training. It may be possible, with slight modifications, to gradually increase the complexity of the environment in contrast to the large jump from static to dynamic environments made in this paper. Another benefit would also be to accelerate the learning process, especially

with the potential Q-learning has in this scenario.

8 Conclusion

In a discrete grid-based Atari Pong static environment, the Q-learning algorithm shows the most potential for generalization to a dynamic environment due to the exploration and continuous improvement overtime. Monte Carlo and SARSA algorithms achieved high winning percentages rapidly, but perfected a very small subset of the state space, limiting their ability to adapt. The performance of these two algorithms appeared to asymptote quickly due to its lack in generalization. In this use case, the weight of exploration proved to be more important than exploitation. These findings suggest that rapid exploitation may show great performance in the short term, long term adaptability is driven by more emphasis on exploration.

References

Brockman, Greg, V. H. Vanhoucke, J. Degris, A. M. Gulcehre, D. Amodei, and S. Leibo. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540,* Available at <https://arxiv.org/abs/1606.01540>.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, and et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (7540):529–533. https://doi.org/10.1038/nature14236.

Modany, Angela. 2012. "Pong, Atari, and the Origins of the Home Video Game". Accessed: 10.09.2024. Available at <https://americanhistory.si.edu/blog/2012/04/pong-atari-and-the-origins-of-the-home-video-game.html>.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction.* 2nd. MIT Press.

Watkins, Christopher J.C.H. 1989. Learning from Delayed Rewards. *PhD Thesis, King's College, University of Cambridge.*