Exploring the Performance of Policy Iteration, Value Iteration, SARSA, and Q-Learning on Discrete and Continuous Problem Spaces

Nico Medellin[1]

I. Abstract

Markov Decision Processes (MDPs) are a mathematical framework used to describe decision-making in situations where outcomes are partly random and partly under the control of an agent. They provide a formal structure for defining real world systems in which an agent may have to navigate. In reinforcement learning, environments like Blackjack and CartPole provide simple playground environments that make them useful for studying MDPs because they highlight the challenges of reinforcement learning across very different domains. In this paper, we explore the performance of Policy Iteration, Value Iterations, SARSA, and Q-Learning on Discrete and Continuous Problem Spaces. We focus on tabular versions of these algorithms, which lay the foundation for more advanced variants like Deep Q-Networks (DQN) or Rainbow DQN.

II. Introduction

In reinforcement learning, environments like Blackjack and CartPole provide simple playground environments that make them useful for studying Markov Decision Processes (MDPs).

III. Methods

Blackjack is a turn-based card game with a discrete state space, which is defined as the following: the player's hand sum, the dealer's visible card, and whether the player has a usable ace. Its action space is also discrete, consisting of two actions: "hit" (where the player takes another card from the deck) or "hold" (the player refuses to draw another card).

The player is rewarded +1 for winning, -1 for losing, and 0 for a draw. No additional reward is given for a natural blackjack.

In contrast, CartPole is a physics-based control problem with a continuous state space, defined by cart's position,velocity, the pole's angle and angular velocity of the pole. The cart can have x-position values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range. The pole angle can be observed between ±24°, but the episode terminates if the pole angle is not in the range ±12°. The simulated episode ends when the pole angle is greater than ±12°, the cart position is greater than ±2.4 or the episode length is greater than 500. The agent scores a point for every frame that these rules are not violated, so you can think of the agent's score to be equal to the episode length.

[1]Georgia Institute of Technology, Department of Computer Science

The action space space for the cart pole problem is discrete, where only a left or right force can be applied to the cart. If the cart is pushed to the left then that is marked as a 0, if the cart is pushed to the right is marked as a 1.

These two environments are compelling case studies for MDPs because they highlight the challenges of reinforcement learning across very different domains. Blackjack exemplifies decision-making in a small, finite MDP with stochastic transitions (due to the random nature of drawing cards from an infinite deck), whereas CartPole presents a high-dimensional, continuous problem. Studying both reveals how algorithm performance is influenced by environment structure, state complexity, and the form of feedback they receive.

A. *What are Markov Decision Processes?*

A Markov Decision Process (MDP) is a mathematical framework used to describe decision-making in situations where outcomes are partly random and partly under the control of an agent. An MDP models how an agent can make a sequence of decisions over time to maximize some notion of cumulative reward. An MDP is made up of states S, actions A, a transition function P(s'ls,a) , and a reward function R(s,a), and a discount factor.

- A state represents all the situations the agent can be in.
- The actions represent the choices the agent can make.
- The transition function defines the probability of moving to a new state s' given the current state and action.
- A reward function tells the agent how much immediate reward they get
- A discount factor is used to determine how much future rewards are valued compared to immediate ones.

MDPs are important because they provide a formal structure for defining real world systems in which an agent may have to navigate. If it is possible to design a proper MDP for a real world problem that accurately captures the problem, then it is possible to attempt to train an agent to navigate this real world problem, for example, teaching robots how to navigate tough terrain or teaching cars how to drive themselves.

B. *Explanation of Algorithms*

To solve these problems, we applied both Value Iteration (VI) and Policy Iteration (PI), which are dynamic programming methods that assume full knowledge of the environment's transition probabilities and rewards. VI alternates between evaluating the current value of each state and updating it based on the best possible action, iteratively improving the value function until convergence. PI, in contrast, alternates

between evaluating a fixed policy and improving it by choosing actions that maximize expected returns, gradually refining the policy itself. These algorithms work well for Blackjack due to its small, discrete state space and known transition model. However, for CartPole, the continuous state space had to be discretized before applying these methods. We used a uniform binning strategy to map the continuous values of position, velocity, angle, and angular velocity into a finite grid, allowing us to approximate CartPole as a discrete MDP.

For learning directly from interaction, we turned to SARSA and Q-Learning, which are model-free temporal difference methods. Both maintain a Q-table representing action-value estimates for each state-action pair. SARSA is an on-policy method that updates values based on the action actually taken, whereas Q-Learning is off-policy and updates based on the maximum value of possible next actions, regardless of the current policy. This subtle difference makes Q-Learning more aggressive in learning optimal policies, while SARSA tends to be more conservative and safer in environments with high variance. Though we primarily focused on tabular versions of these algorithms, they lay the foundation for more advanced variants like Deep Q-Networks (DQN) or Rainbow DQN, which scale to complex, high-dimensional environments using neural networks.

1) Value Iteration: Value Iteration is a dynamic programming method used to compute the optimal policy for an MDP when the transition model and reward function are known. It works by iteratively updating the value of each state, which represents the maximum expected cumulative reward that can be obtained starting from that state.

The key idea behind VI is the Bellman optimality equation:

$$V(s) \leftarrow \max_a \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V(s')] \quad (1)$$

At each step, the algorithm:

1) Evaluates the expected utility of each possible action from the current state.
2) Updates the value of the state based on the best possible action.
3) Repeat this process until the value function converges (i.e., changes between iterations fall below a small threshold).

Once the values are stable, the optimal policy π*can be derived by selecting the action in each state that produces the highest expected value.

2) Policy Iteration: Policy Iteration also aims to find the optimal policy, but it does so by alternating between two steps: policy evaluation and policy improvement.

1) Given a fixed policy π, compute the value function $V^\pi(s)$ for all states using:

$$V^\pi(s) = \sum_{s'} P(s' \mid s, \pi(s)) \left[R(s, \pi(s), s') + \gamma V^\pi(s')\right] \quad (2)$$

This step solves a system of linear equations until the value function stabilizes.

2) Update the policy by choosing the action in each state that maximizes the expected value:

$$\pi'(s) = \arg\max_a \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V_\pi(s')] \quad (3)$$

This process is repeated until the policy no longer changes.

3) SARSA (State-Action-Reward-State-Action): SARSA is an on-policy learning algorithm used in situations where the environment model is unknown. Rather than computing the value function using a known transition model, SARSA learns from actual experience gathered by interacting with the environment.

The equation for updating SARSA is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right] \quad (4)$$

where

- s, a: current state and action
- r: reward received
- s',a': next state and next action chosen using the current policy (often ϵ-greedy)
- α: learning rate
- γ: discount factor

Because SARSA update the Q-value values based on the action taken, it tends to do well in environments with high levels of uncertainty.

For our experiments, we are using a Standard Tabular SARSA with a ϵ-greedy exploration that follows an annealing schedule. We are using this form of the algorithm because it is easy to understand and serves as a great starting point for understanding the SARSA algorithm. We use epsilon to update how we choose the next action in an iteration. Epsilon is the measure of the probability of taking a random action in the next iteration for a given state, if the value is set to 1, then there is a 100% chance that a random action will be taken.

4) Q-Learning: Q-Learning is an off-policy algorithm, meaning it learns the optimal policy regardless of the agent's current behavior policy. Its update rule is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \quad (5)$$

The main difference between SARSA and Q-Learning is that Q-Learning uses the maximum estimated future reward, rather than using the value of the next action taken. Q-Learning is considered to be more aggressive in seeking out optimal strategies and will often converge to optimal policies faster, especially if the environment is deterministic.

For our analysis of our MDPs, we used basic tabular Q-learning using ε-greedy exploration algorithm with an

annealing schedule for learning and exploration rates. Depending on the initial value of ε (the closer to 1), the more our algorithm will tend toward exploration. Epsilon is the measure of the probability of taking a random action in the next iteration for a given state, if the value is set to 1, then there is a 100% chance that a random action will be taken.

5) Discretizations of the Cart Pole Problem: Given that the cart pole simulation exists in a continuous environment, VI, PI, SARSA, and Q-learning are unable to generate policies for a continuous environment. Therefore, we must discretize the environment so we can attempt to simulate it. For the purpose of our study, we are going to represent the cart pole problem in two different ways. The first method is that we are going to break down cart position, cart velocity, and pole angular velocity into 10 uniform bins, with angular center resolution and angular outer resolution set to 0.1. For our second environment, we are keeping the angular center resolution and angular outer resolution the same, but we are breaking up the cart position, velocity, and angular velocity into 5 uniform bins.

Given that the cart pole is a continuous problem space, the closer and closer the number of bins approaches infinity, the closer we get to simulating the continuous problem space. For example, 100 bins does a much better job of approximately the original problem space than having 10 bins does. However, attempting to simulate more bins leads to a significant increase in the amount of computational resources and time required to simulate the original environment and it drastically increases the amount of time required to run solutions like SARSA and Q-learning. Increasing the number of bins also increases the amount of states that needs to be simualted and developed a policy for. Setting up a cart pole problem with 50 bins and running PI on the environment led to a policy with more than 2.88 million entries, where the policy for 10 bin cart pole is only several thousand entries.

IV. Results

A. Blackjack

Analyzing the results (Figure 1 and 2) for blackjack we see that PI and VI convergence very quickly (within 10 iterations). This is most likely due to the simple nature of the blackjack problem along with the simple nature of the VI and PI algorithm. Looking at figure 3 and figure 5, we see that SARSA and Q-learning take much longer to converge (typically around the 500 episode mark), this is due to the fact that they are model free.

From a mean value perspective, it looks like VI and PI outperform SARSA and Q Learning. This is to be expected given that PI in theory for any finite MDP with known dynamics should converge to the optimal policy.

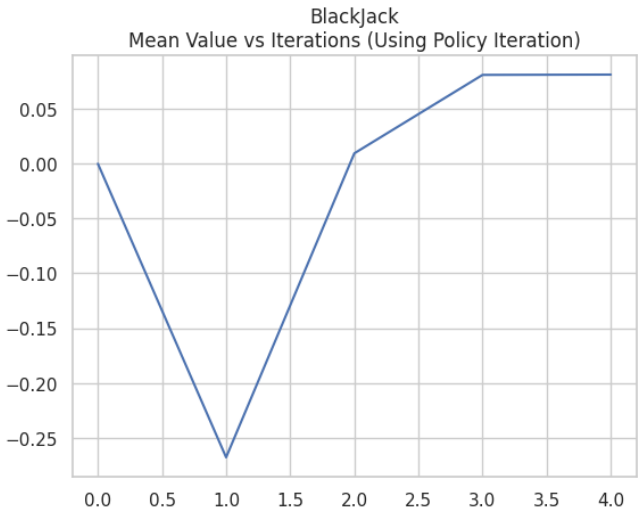

Figure 1 *Blackjack Mean Value vs Iterations using Policy Iterations*

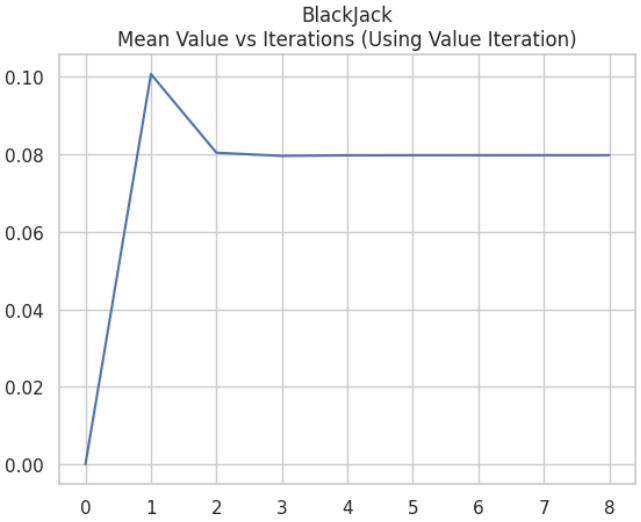

Figure 2 *Blackjack Mean Value vs Iterations using Value Iterations*

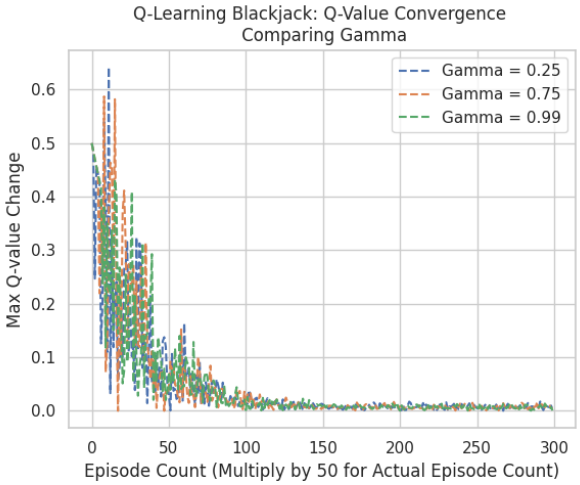

Figure 3 *Q Learning Convergence of Blackjack*

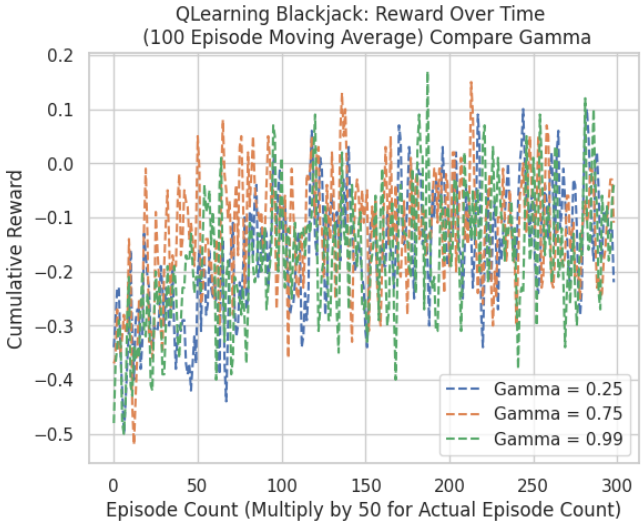

Figure 4 *Q Learning Reward Over Time for Blackjack*

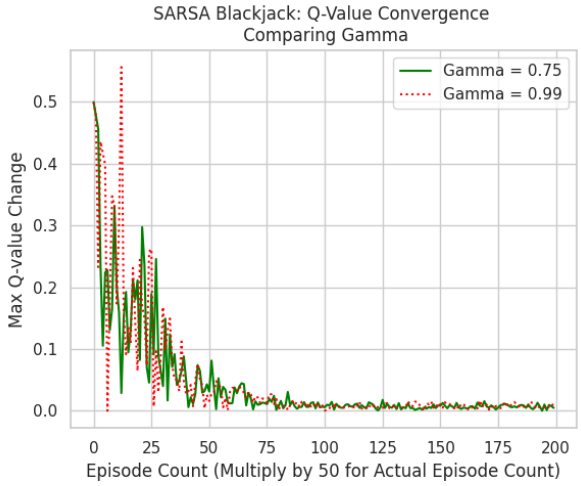

Figure 5 *Q Value Convergence of Blackjack Using SARSA*

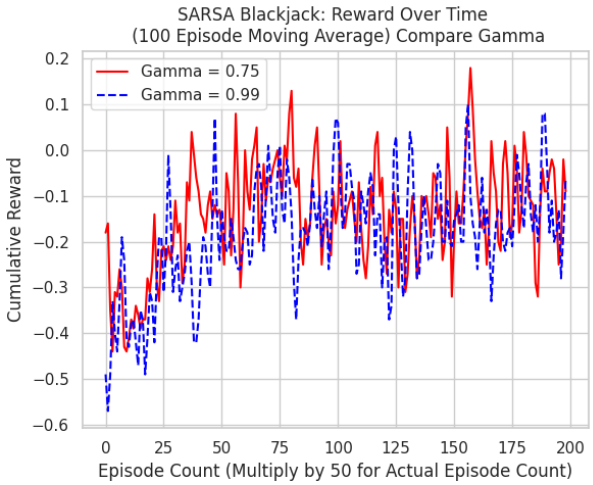

Figure 6 *100 Episode Moving Average of Blackjack Using SARSA*

Another interesting takewawy from the blackjack problems is the fact that the policy maps are slightly different for all the models (PI, SARSA, and Q-Learning) (Figures 7, 8, 9). Specially the fact that even though SARSA and Q-learning have very similar rewards over time, they still have differences in the specific strategies given a certain hand. For future work, I would want to compare each cell of the policy map for each of these algorithms with the optimized and solved solution for blackjack, in order to calculate which of the policies is the most accurate when compared to the optimal solution of blackjack. Given the fact that Q Learning and SARSA had similiar performances yet they have different polocies, this leads me to believe that both policies may deviate from the optimal solution by a similar amount.

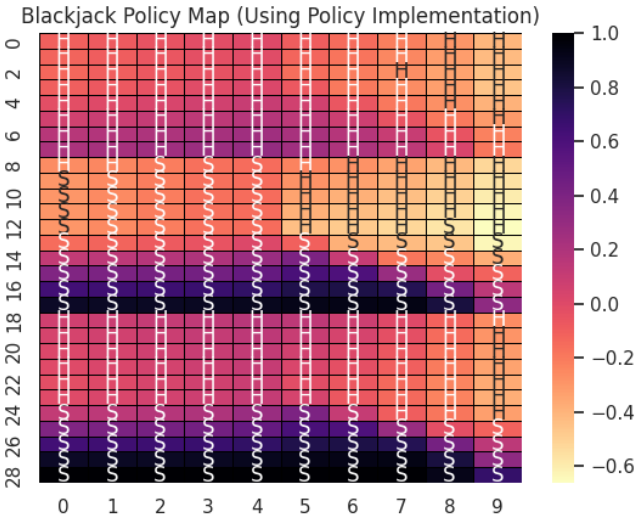

Figure 7 *Policy Map Using PI*

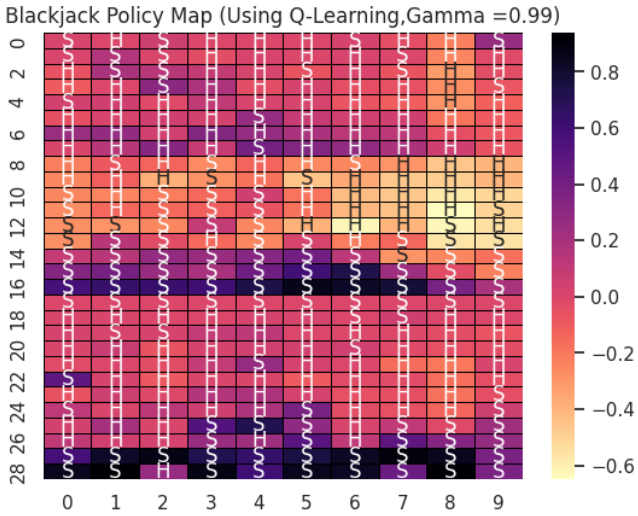

Figure 8 *Policy Map Using Q Learning*

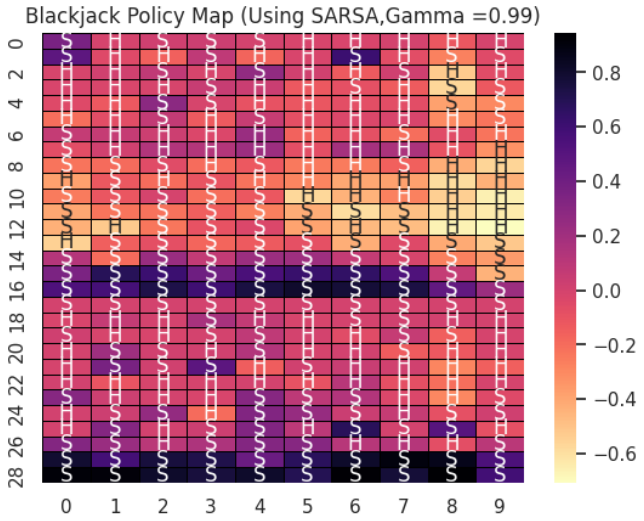

Figure 9 *Policy Map Using SARSA*

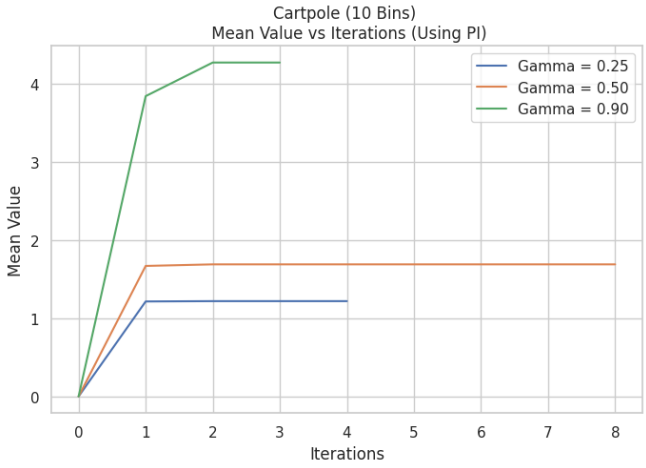

Figure 11 *The Effect of Gamma on PI for Cartpole 10 Bin*

B. Cart Pole

Across the board we see that higher levels of gamma lead to a better outcome for VI, PI, SARSA, and Q-Learning. This is expected for VI and PI as it controls the how much the algorithm prioritizes long term rewards. The higher the gamma, the more focus on long term results. This also applies to SARSA and Q-Learning. This makes sense for this type of problem as the score increases the longer you keep the game running. For example, if the model were to try and focus on solely getting to a low score of say 10, over its long term goal of trying to get to 100, then our model would get stuck in at a local minimum. You can see the better performance by the higher gammas in vigure 10 and 11 due to the higher mean value. You can also see it in figure 13 and 14 by the significantly higher cumulative rewards when compared to the lower gammas.

In figures 13 and figure 15, we do see that lower gammas do converge faster for SARSA and Q-learning, however, this is at the expense of not finding an optimal solution.

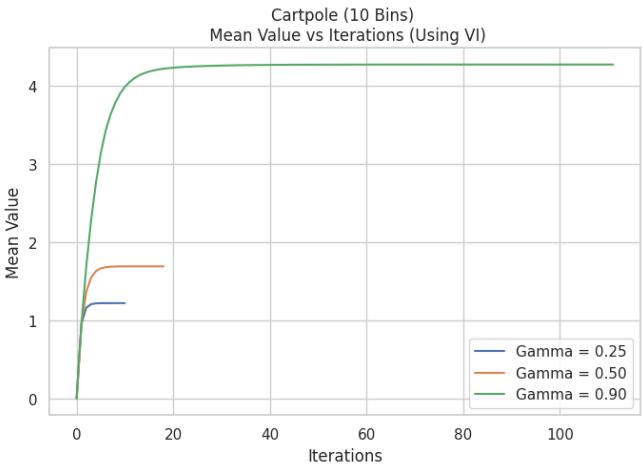

Figure 10 *The Effect of Gamma on VI for Cartpole 10 Bin*

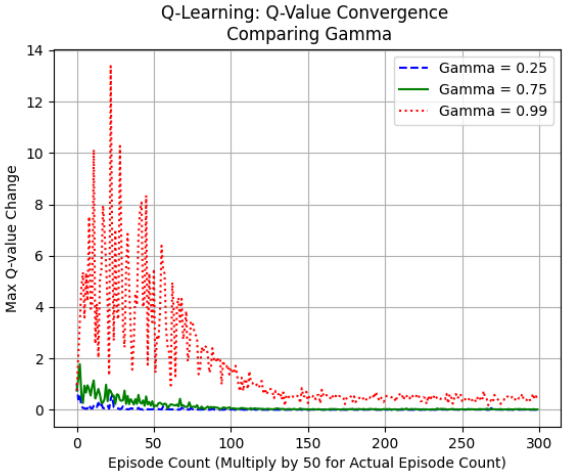

Figure 12 *The Effect of Gamma on Q-Learning Convergence for Cartpole 10 Bin*

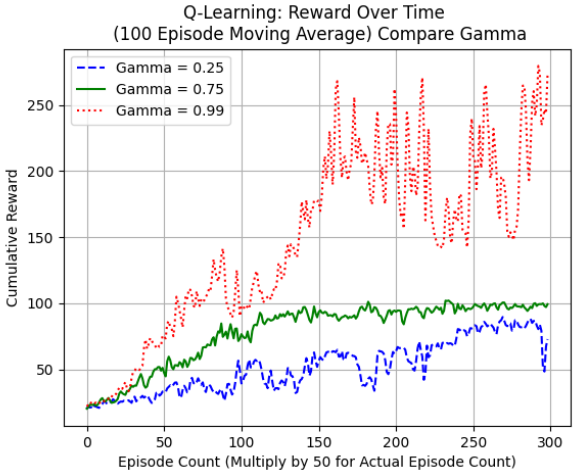

Figure 13 *The Effect of Gamma on Q-Learning Reward Over Time for Cartpole 10 Bin*

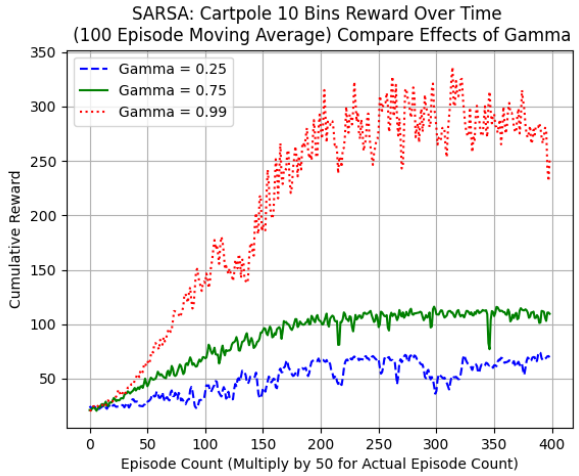

Figure 14 *The Effect of Gamma on SARSA Reward Over Time for Cartpole 10 Bin*

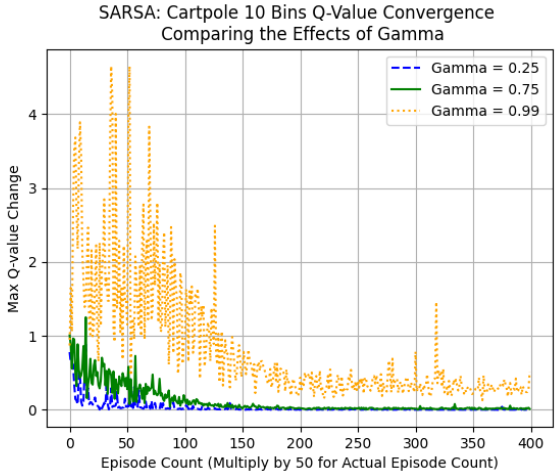

Figure 14 *The Effect of Gamma on SARSA Convergence for Cartpole 10 Bin*