# Lab Report: Reinforcement Learning Algorithms for CartPole-v0

Chirayu Salgarkar

October 16, 2024

**Abstract**

This report explores and implements various reinforcement learning algorithms to solve the CartPole-v0 environment from OpenAI Gym. The methods include tabular approaches like SARSA and Q-Learning using state-space discretization, and function approximation techniques like linear approximation and deep Q-networks (DQN). The performance of these algorithms is evaluated and compared based on learning speed and policy quality.

## Contents

# 1 Introduction

The CartPole-v0 environment presents a control problem where the goal is to balance a pole on a moving cart by applying discrete forces. The environment has four continuous state variables: cart position, cart velocity, pole angle, and pole tip velocity, with ranges of $[-4.8, 4.8], [-\infty, \infty], [-24°, 24°]$, and $[-\infty, \infty]$, respectively.

Our goals in this exercise were:

- Use tabular techniques to discretize a continuous state space
- Implement and compare SARSA and Q-Learning
- Use and develop function approximation techniques, including linear function approximation and DQN

This report evaluates the efficacy of these methods based on performance metrics like convergence speed and final policy quality.

# 2 Methods and Implementation Results

## 2.1 State Space Discretization

CartPole's continuous state space needs to be discretized for tabular methods like SARSA and Q-Learning. We divided each of the four state variables into bins to convert the continuous observations into discrete states. The number of bins was chosen to be 15 to balance the trade-off between computational feasibility and granularity.

Two techniques were analyzed: Uniform Binning, and Adaptive binning (sigmoid bins). Uniform binning refers to a digitization of state values into bins after arbitrary policy selection. On the other hand, sigmoid binning uses a sigmoid function ($\frac{e^x}{1+e^x}$) or hyperbolic tangent to bin based on the $x$ value, as it then allows for more granularity near the origin.

### 2.1.1 Discretization Pseudocode

Here is a simplified pseudocode for the binning process:

Listing 1: Discretization Algorithm

```
def discretize_state(state, bins, state_bounds):
    state_index = []
    for i in range(len(state)):
        # Clip the state to be within bounds
        clipped_state = min(max(state[i], state_bounds[
            i][0]), state_bounds[i][1])
        # Find the bin index for each state variable
        bin_index = np.digitize(clipped_state, bins[i])
            - 1
        state_index.append(bin_index)
    return tuple(state_index)
```

Ultimately, uniform binning was chosen due to ease of implementation, as seen below:

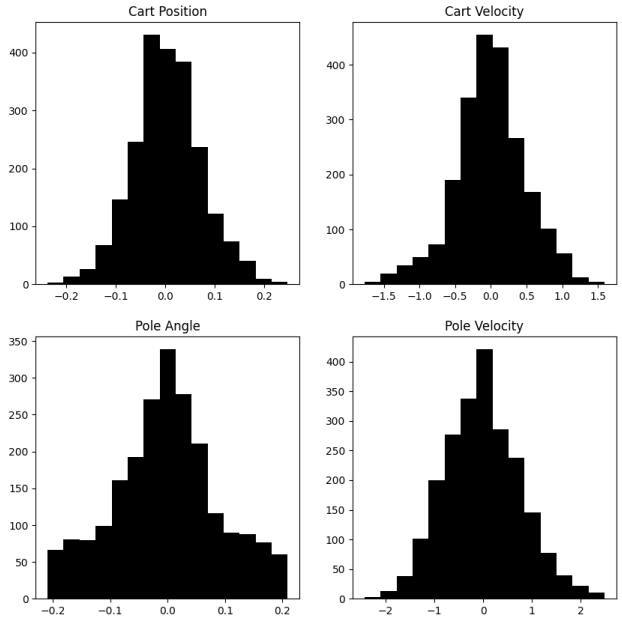


Figure 1: Demonstration of state observations after binning put in place.

Further efforts should incorporate adaptive or sigmoidal binning based on the distribution, especially if the setup is brought into real-world experimentation.

## 2.2 TD Learning - SARSA and Q-Learning

TD learning is tabular - TD methods learn action values in a Q-table. SARSA is an on-policy method, meaning it updates the Q-values based on the action actually taken by the agent according to its current policy. Q-learning is off-policy, meaning it updates the Q-values assuming the agent always chooses the optimal action in the future, regardless of its current policy.

## 2.3 SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is an on-policy temporal difference learning algorithm that estimates the action-value function based on current policy actions. Using the discretized state space, we implemented SARSA to learn an optimal policy for the CartPole environment. The up-

date rule is given as follows:

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

### 2.3.1 SARSA Implementation

SARSA led to a convergence of about 175 which is slightly suboptimal over the episode link. Hyperparameter tuning at that state required epsilon values of 0.45, an extremely high decay value 0.9998, and a relatively standard learning rate of 0.1. Program termination was on the order of 20 minutes.

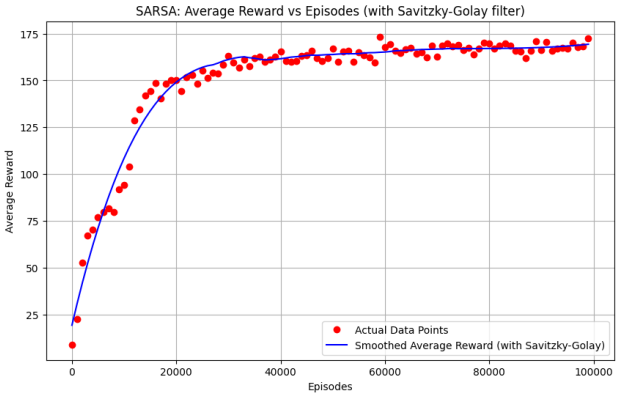


Figure 2: SARSA graph shows an increase in reward with respect to time. Graph shows both raw data and data through a Savitsky-Golay filter.

## 2.4 Q-Learning Algorithm

Q-Learning, on the other hand, is an off-policy temporal difference learning algorithm, which acts greedily, updating the action-value function based on optimal possible actions in the future. The update rule is given as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[ R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{2}$$

### 2.4.1 Q-Learning Implementation

Under the same hyperparameters as those given in SARSA, optimal policy convergence occurred slightly faster, at 35000 episodes, as seen in the figure.

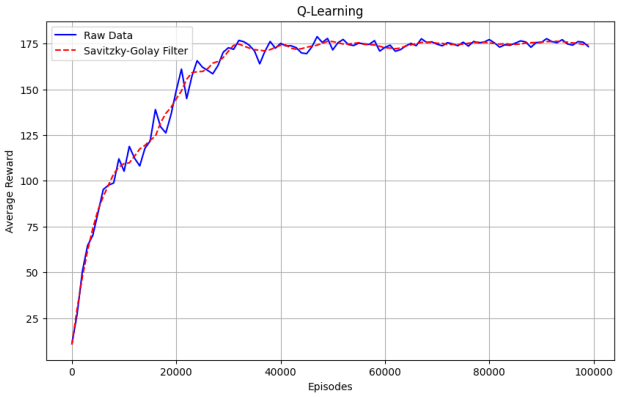


Figure 3: Q-learning graph shows policy convergence in fewer episodes as SARSA. Graph shows both raw data and data through a Savitsky-Golay filter.

The advantage of Q-Learning is evident, as its off-policy nature allows it to exploit future optimal actions more effectively.

## 2.5 Performance Comparison: SARSA vs. Q-Learning

Table 1 compares the performance of SARSA and Q-Learning, focusing on key metrics like convergence speed and stability.

| **Metric** | **SARSA** | **Q-Learning** |
|---|---|---|
| Convergence Time (steps) | 50,000 episodes | 35,000 steps |
| Exploration Strategy | On-policy | Off-policy |
| Policy Stability | Moderate | Moderate |
| Computation Time | 20 mins | 15 mins |

Table 1: Performance comparison between SARSA and Q-Learning.

## 2.6 Function Approximation

For non discretized state-spaces, function approximation is generally used. Two function approximation methods were implemented: Linear Function Approximation and Deep Q-Learning (DQN).

## 2.7 Linear Function Approximation

In function approximation, the Q-function is represented as a linear combination of features derived from the state variables. This allows us to handle larger or continuous state spaces efficiently. For this task, Linear Function approximation was conducted for the four state values under the feature vectors. Weights end up updating through gradient descent. The stochastic gradient descent update rule is effectively the step size multiplied by the prediction error multiplied by the feature value.

### 2.7.1 Linear Function Approximation Pseudocode

Listing 2: Linear Function Approximation Update

```
def update_weights(theta, features, target, alpha):
    prediction = np.dot(theta, features)
    error = target - prediction
    theta += alpha * error * features
    return theta
```

### 2.7.2 Linear Function Approximation implementation

An extremely low learning rate was needed for parametrizing both linear function approximation and DQN. Epsilon values were at 0.9, with decay being 0.9995. The performance of the data is shown in the figure below.

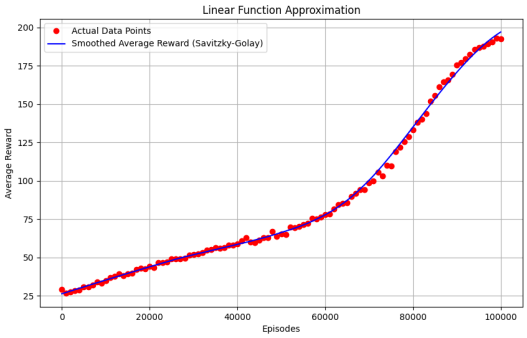


Figure 4: Linear Function Approximation graph shows policy convergence over a longer period of time, albeit with far greater hyperparameter tuning required. Graph shows both raw data and data through a Savitsky-Golay filter.

## 2.8 Deep Q Network (DQN)

DQN is an extension of Q-learning that uses a deep neural network to approximate the action-value function $Q(s, a)$. It handles large state spaces by learning from experience using the following key components:

1. **Q-Network:** Approximates $Q(s, a; \theta)$, where $\theta$ represents the network weights.
2. **Experience Replay:** Stores transitions $(s, a, r, s')$ and samples mini-batches for training.
3. **Target Network:** A separate network $Q_{\text{target}}(s, a; \theta^-)$ to provide stable target values.
4. **Loss:** Minimize the squared difference between the predicted Q-value and the target.

### 2.8.1 DQN Algorithm

1. Initialize $Q(s, a; \theta)$ and $Q_{\text{target}}(s, a; \theta^-)$
2. For each episode:
   (a) Initialize state $s$

(b) For each step $t$:

- Select action $a_t = \arg\max_a Q(s_t, a; \theta)$ (with $\epsilon$-greedy exploration)
- Observe reward $r_t$ and next state $s_{t+1}$
- Store $(s_t, a_t, r_t, s_{t+1})$ in the replay buffer
- Sample mini-batch from the buffer, compute targets:

$$y_j = r_j + \gamma \max_{a'} Q_{\text{target}}(s_{j+1}, a'; \theta^-)$$

- Perform a gradient descent step on $L(\theta)$

(c) Every $C$ steps, update target network: $\theta^- \leftarrow \theta$

### 2.8.2 DQN Implementation

A neural net was generated for Q-value approximation outside of the submission database. However, using the same hyperparameters as Linear Function Approximation, it was still very difficult to properly train the DQN implementation as seen below.

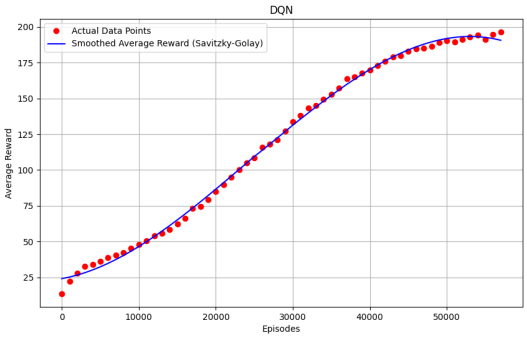


Figure 5: DQN graph shows slow convergence, requiring significant parameter tuning. Graph shows both raw data and data through a Savitsky-Golay filter.

# 3 Discussion

The performance of different reinforcement learning (RL) methods varied significantly in terms of convergence speed and stability. Q-learning showed

faster convergence compared to SARSA, likely due to its off-policy nature. Function approximation methods, particularly DQN, were highly sensitive to hyperparameter tuning, especially the learning rate. This sensitivity is expected since function approximators, like neural networks, can lead to divergence when parameters are improperly tuned.

SARSA's on-policy nature made it more stable in environments with stochastic elements, but it often converged to sub-optimal policies. Q-learning, benefiting from its off-policy updates, exploited future actions more effectively, resulting in faster convergence. However, Q-learning's instability was evident in noisy environments, where the learned policy might deviate significantly from the behavior policy. Linear function approximation handled continuous state spaces without discretization but required careful learning rate tuning to avoid overshooting.

DQN had the potential for better function approximation in continuous state spaces but was particularly sensitive to hyperparameters. While DQN can learn more complex policies, the noisy and unstable training process posed significant challenges, requiring more careful tuning and computational resources. In resource-constrained settings, such as this assignment, these limitations became more pronounced, preventing fully optimized results.

The primary challenges encountered were related to hyperparameter tuning and computational expense. Finding optimal parameters, especially for DQN, was impractical within the time constraints. Additionally, implementing RL algorithms in practice was more complex than the theory suggested. For example, Q-learning and SARSA, despite their simplicity in equation form, required extensive code to implement and train effectively. The lack of a transition matrix in the CartPole environment also made policy iteration challenging, forcing reliance on approximations like Q-learning.

# 4 Conclusion

In this lab report, we implemented various reinforcement learning algorithms, including SARSA, Q-Learning, linear function approximation, and DQN, to solve the CartPole-v0 problem. We found that Q-Learning exhibited faster convergence than SARSA, and that function approximation methods required significant effort in hyperparemeter tuning for optimizing results.

This project proves valuable for real-world implementation. In the real world, while theoretically a function approximation method would be valu-

able for its ability to handle convergence without immediate discretization, it is difficult to implement in practice, especially without signiificant computational resources for implementations. Further work would likely be to improve hyperparameter tuning as well as determine a framework for choice of method depending on computational resources, time, and accuracy requirements in the real world.

# 5 References


- Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction.*
- OpenAI Gym documentation: `https://www.gymlibrary.dev/environments/classic_control/cart_pole/`