

# **BLACKJACK AUTOMATON USING REINFORCEMENT LEARNING: CSPE65 MACHINE LEARNING AND TECHNIQUES PROJECT**

DONE BY:

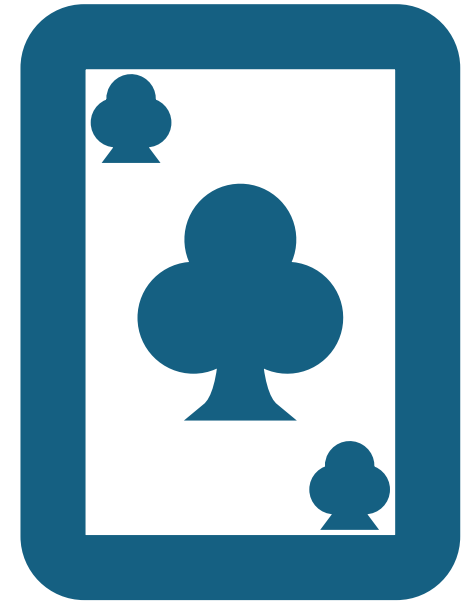
106121082 – Naveen Suresh Nair

106121092 – Prajwal Sundar



# What is BlackJack?

Blackjack is a popular card game played in casinos worldwide. The objective is to beat the dealer's hand without exceeding a total of 21 points. Players aim to achieve a higher score than the dealer or have the dealer bust while avoiding going over 21 themselves.



# Game Rules - 1

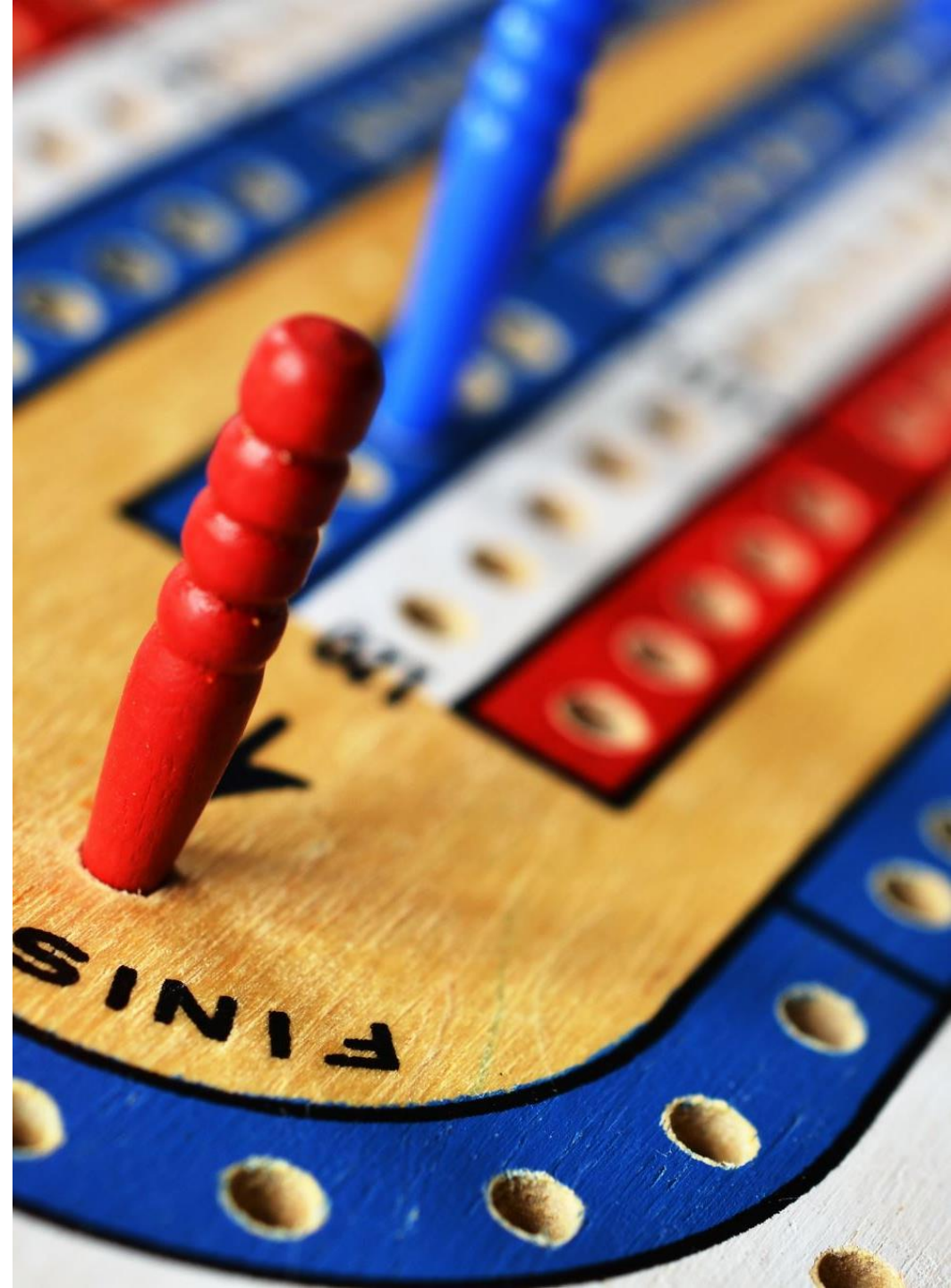
- *Objective:* The objective of Blackjack is to beat the dealer's hand without going over 21.
- *Card Values:* Cards 2 through 10 are worth their face value. Face cards (Jack, Queen, King) are each worth 10 points. Aces can be worth either 1 or 11 points, depending on which value benefits the hand more.
- *Initial Deal:* Each player is dealt two cards face up, while the dealer receives one card face up and one card face down.



# Game Rules - 2

## Player Actions:

1. *Hit*: Players can request additional cards to improve their hand total.
2. *Stand*: Players can choose to keep their current hand and not receive any more cards.





## Game Rules - 3

- *Dealer Actions:* The dealer must hit until their hand reaches a total of 17 or more. If the dealer busts (exceeds 21), all remaining players win.
- *Winning:* Players win if their hand total is higher than the dealer's without going over 21, or if the dealer busts. A "Blackjack" is a hand totaling 21 with the first two cards (an Ace and a 10-value card), which typically pays out at a higher rate.

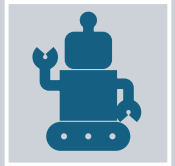




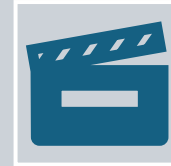
# Basic Approach

- Player would hit until sum is at-least 12.
- After that we use RL model to predict whether to hit or stand
- Based on the results we update the q-Values according to the policies
- We repeat this process until we converge values

# MAPPINGS



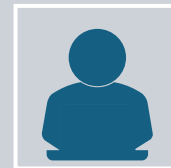
State = [PlayerSum,  
DealerSum, HasAce]



Action = [Hit, Stand]




Reward = [ win:1,  
draw:0, lose:-1 ]



Policy: Based on the  
algorithm

# REINFORCEMENT MODELS USED



Q-  
Learning

Sarsa

Monte  
Carlo



# Q-LEARNING: OFF-POLICY RL

- Q-learning is considered an off-policy reinforcement learning (RL) model because it learns the value of the optimal policy independently of the agent's behavior policy.
- In off-policy learning, the agent learns the value of the optimal policy (the policy that maximizes expected cumulative rewards) while following a different behavior policy (the policy used to interact with the environment).
- Q-learning updates its Q-values based on the maximum Q-value of the next state, regardless of the action taken by the behavior policy. This independence allows Q-learning to learn from experiences collected under any exploratory behavior.





# Unique Features of Q-Learning

---

- *Stability*: Q-learning tends to be more stable compared to on-policy methods because it updates Q-values based on the maximum Q-value of the next state rather than directly using the value of the next state-action pair.
- *Off-Policy Learning*: As an off-policy method, Q-learning can learn from a mix of exploratory and greedy actions, making it more flexible in exploration.
- *Sample Efficiency*: Since Q-learning can learn from past experiences regardless of the behavior policy, it can potentially reuse data more efficiently than on-policy methods.
- *Flexibility*: Q-learning can be applied to a wide range of RL problems and environments without significant modification, making it a versatile algorithm.

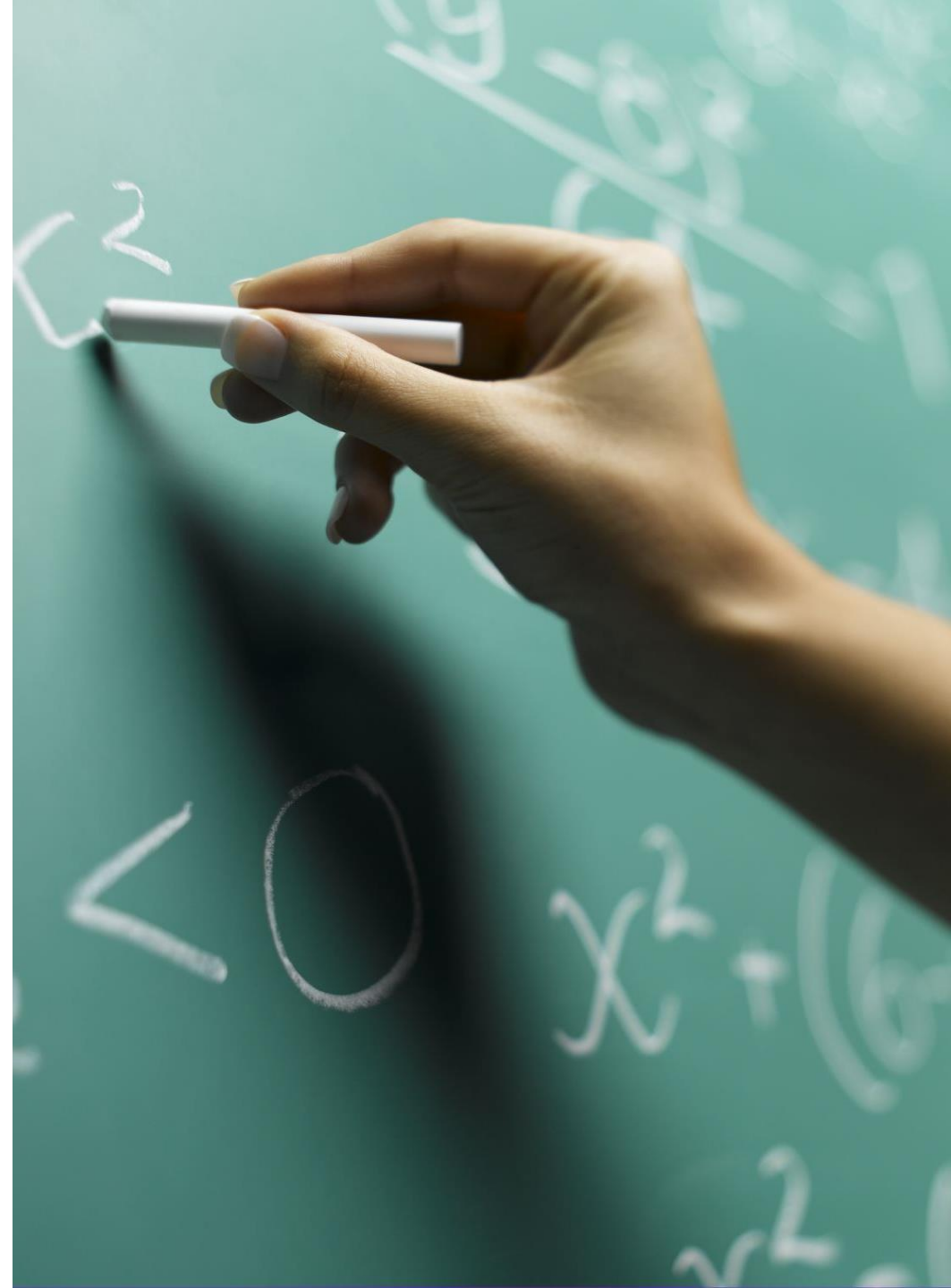
# Formula used in Q-Learning

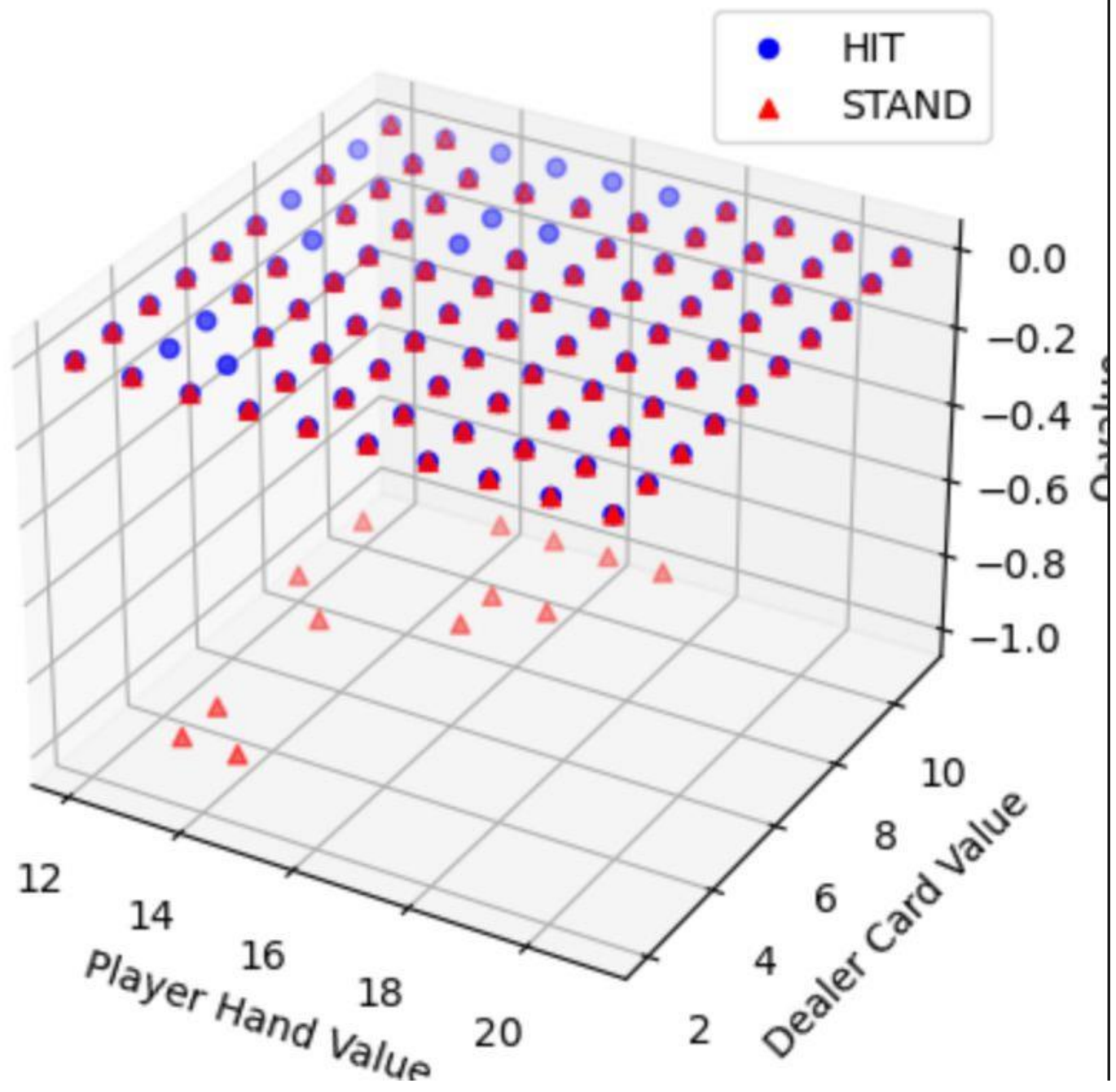
- *Q-Value Update Rule*: The Q-value for a state-action pair is updated iteratively using the Bellman equation:

$$Q(s, a) = Q(s, a) + \alpha * (\text{reward} - Q(s, a))$$

where:

- $Q(s, a)$  is the Q-value for state 's' and action 'a'.
- $\alpha$  (alpha) is the learning rate, determining the step size of the update.
- reward is the immediate reward received after taking action 'a' in state 's'







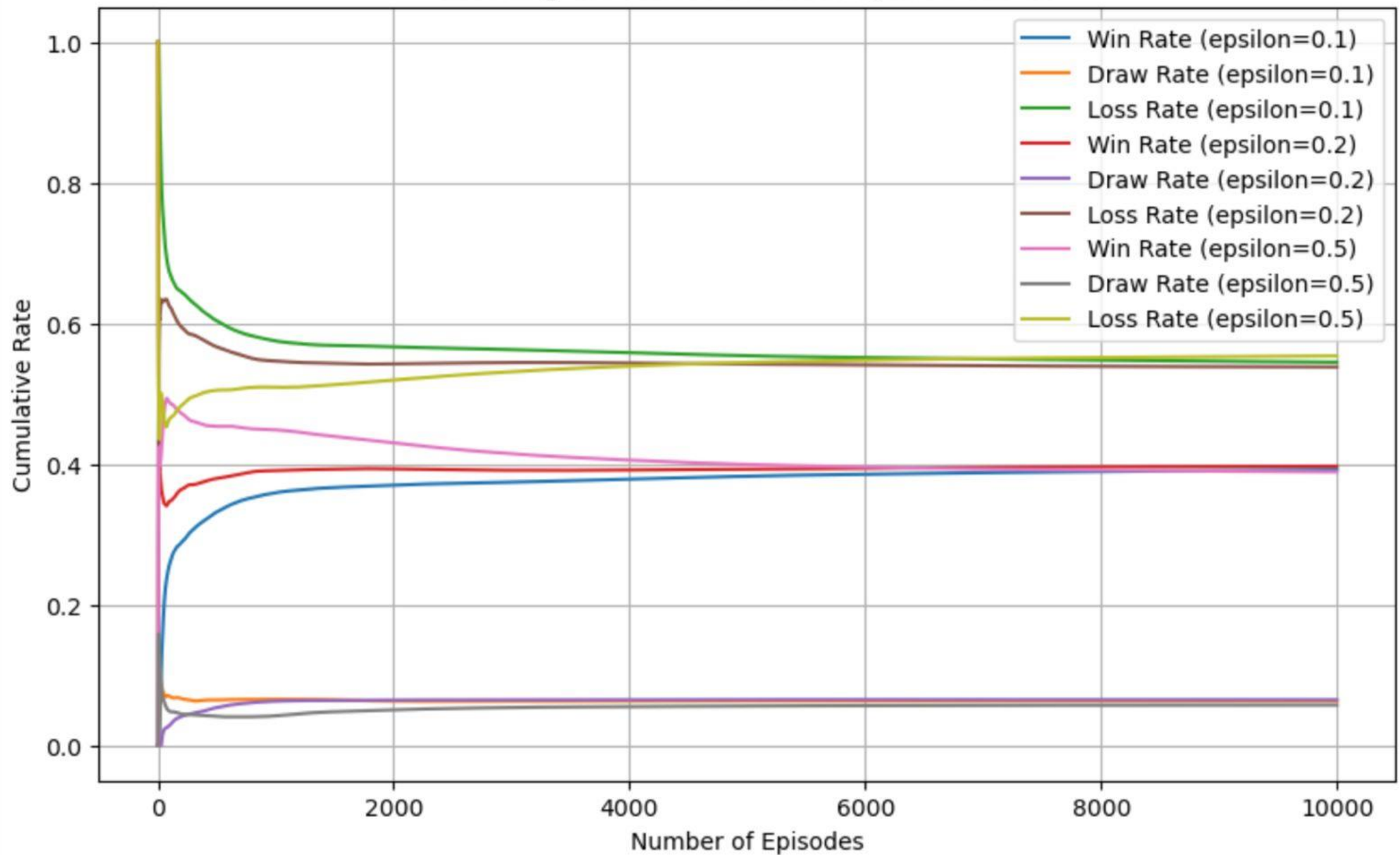
### With a usable Ace:

[illegible]

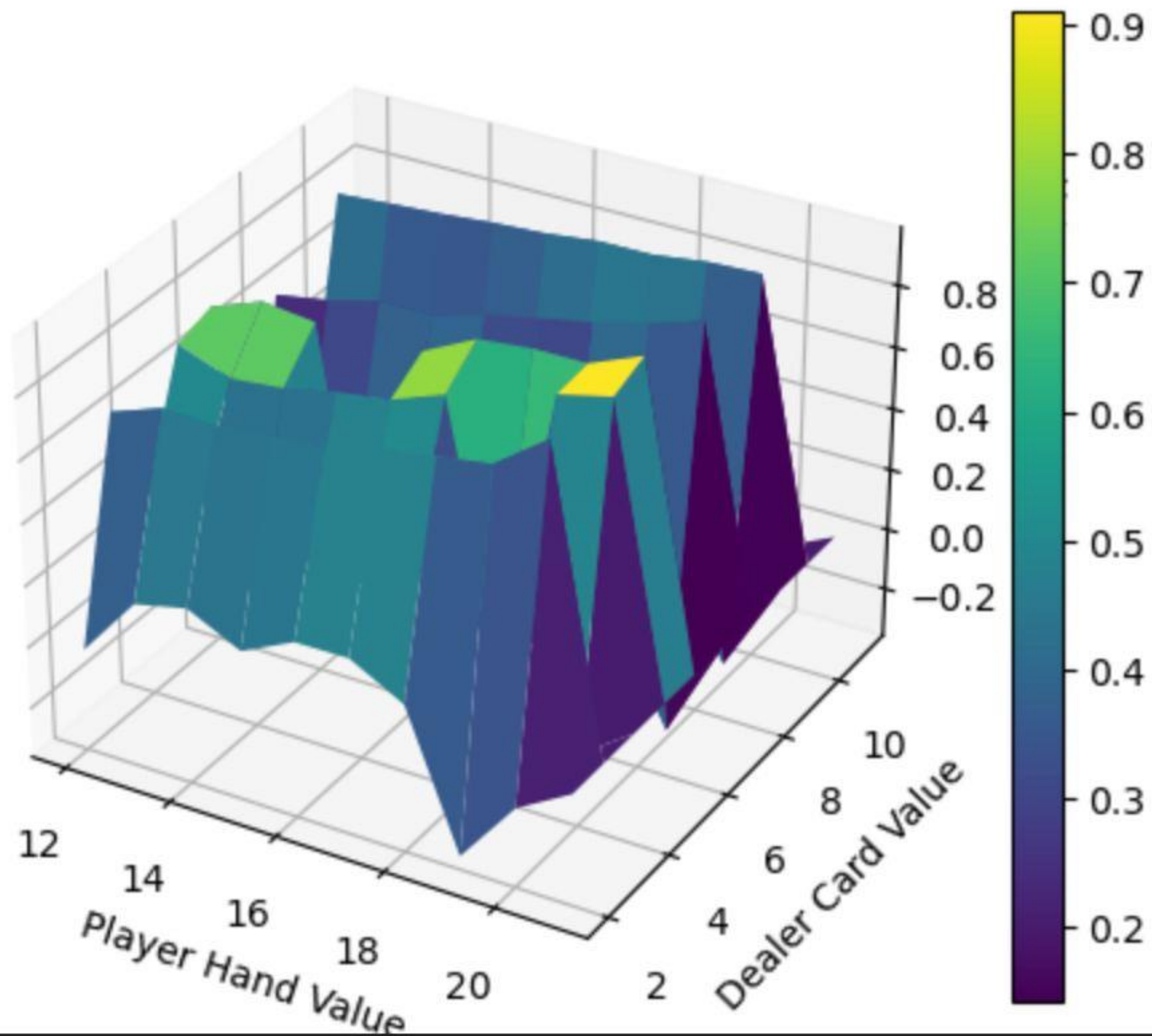
### Without a usable Ace:

[illegible]

Learning Curves for Different Epsilon Values

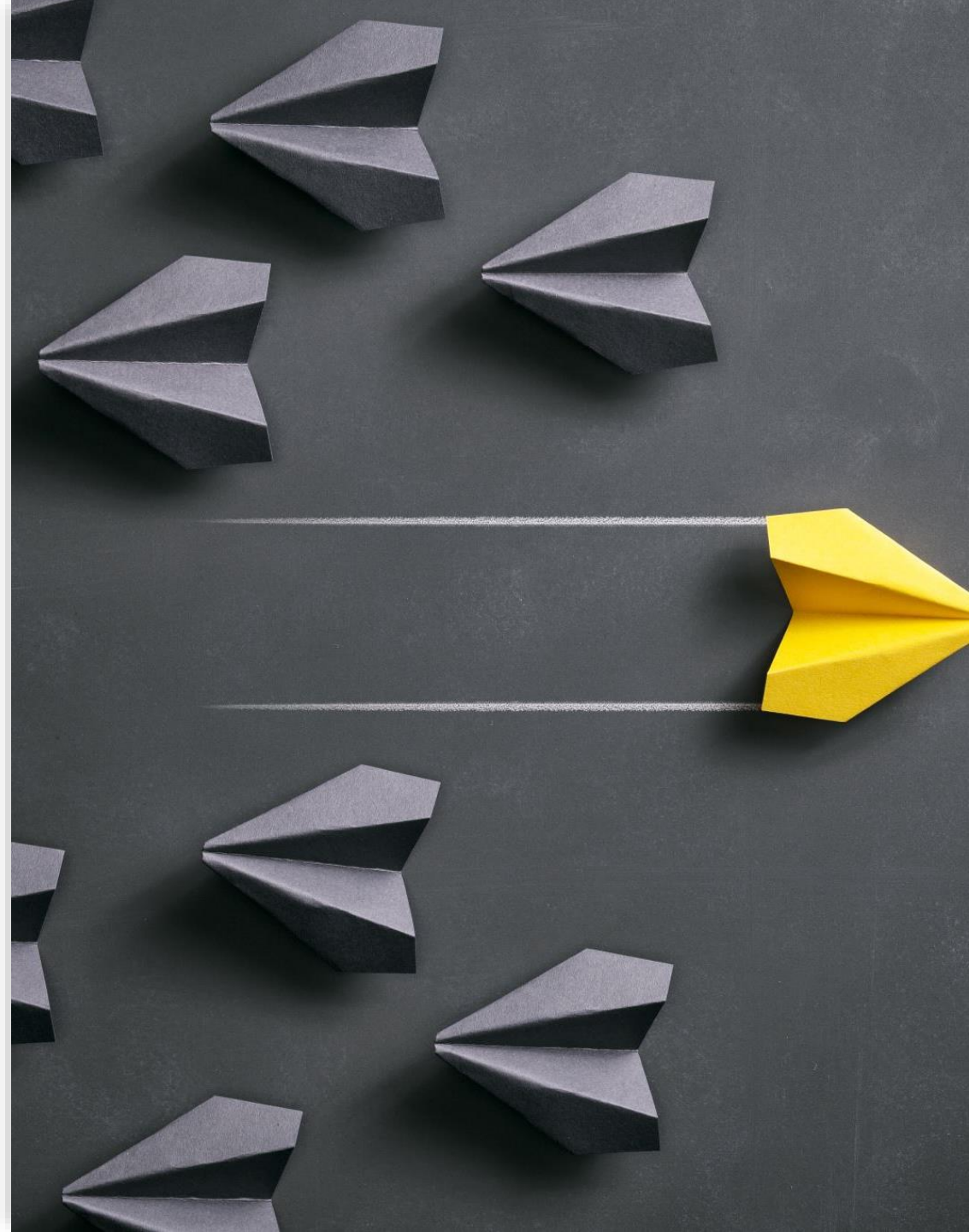




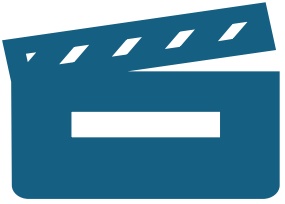


# SARSA ALGORITHM

- SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm used for learning the optimal policy in a Markov Decision Process (MDP).
- SARSA is a model-free method that learns directly from experience without explicitly modeling the transition probabilities or the reward function of the environment.
- It belongs to the class of Temporal Difference (TD) learning algorithms, which update estimates based on the difference between current and future estimates.



# SARSA: ON-POLICY RL



## State-Action Value Function:

SARSA learns the Q-value (state-action value function), denoted as  $Q(s, a)$ , which represents the expected cumulative reward of taking action 'a' in state 's' and following a specific policy thereafter.

The Q-values are updated iteratively based on observed transitions and rewards.



## On-Policy Learning:

SARSA learns the value of the policy being followed. It updates its Q-values based on experiences generated by following the same policy used to select actions (i.e., the behavior policy and the target policy are the same).

This on-policy characteristic makes SARSA sensitive to the exploration strategy of the current policy and can lead to slower convergence if exploration is insufficient.



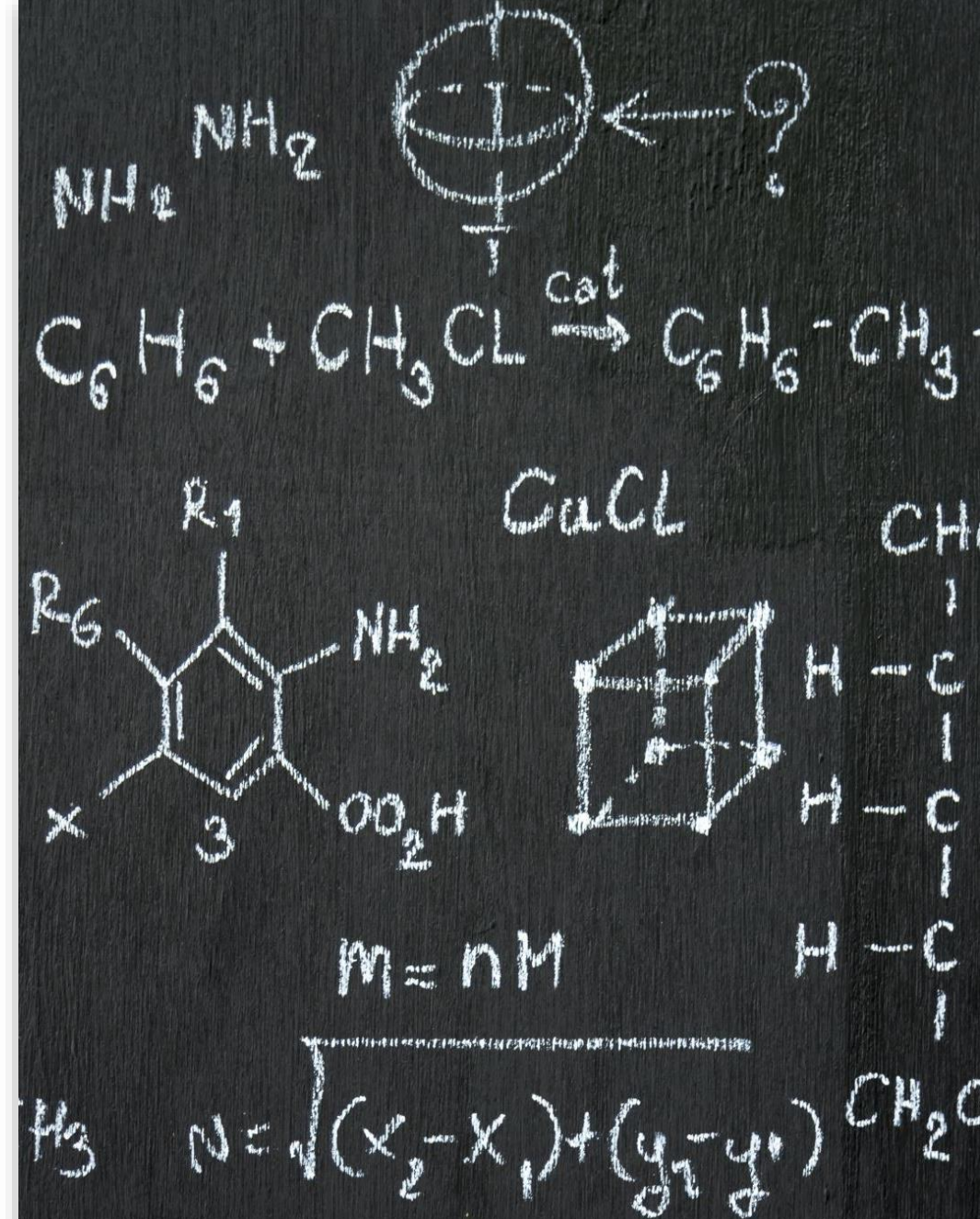
# Formula used in Sarsa Algorithm

The SARSA update rule is based on the Bellman equation for Q-values:

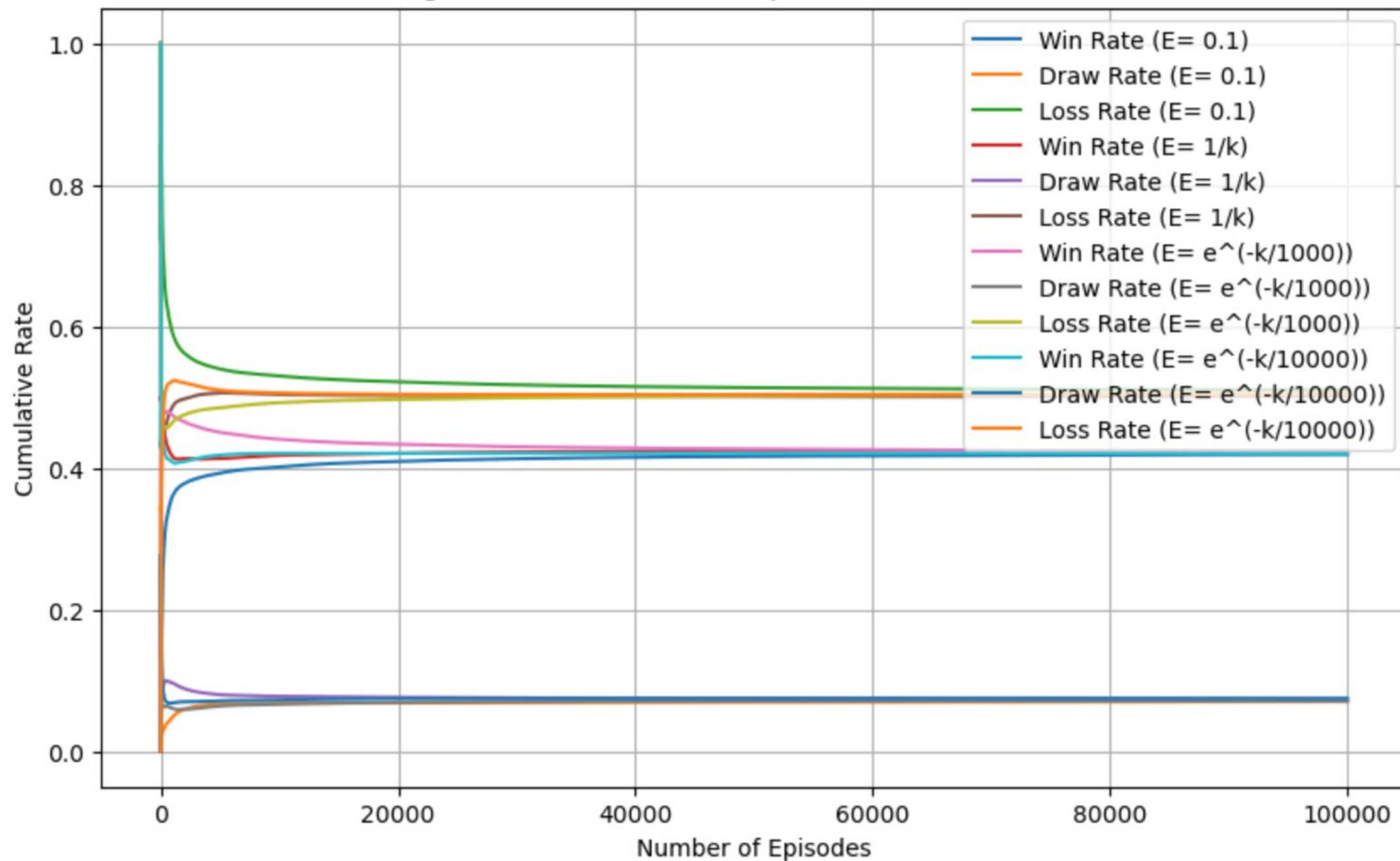
$$Q(s, a) = Q(s, a) + \alpha * (\text{reward} + \gamma * Q(s', a') - Q(s, a))$$

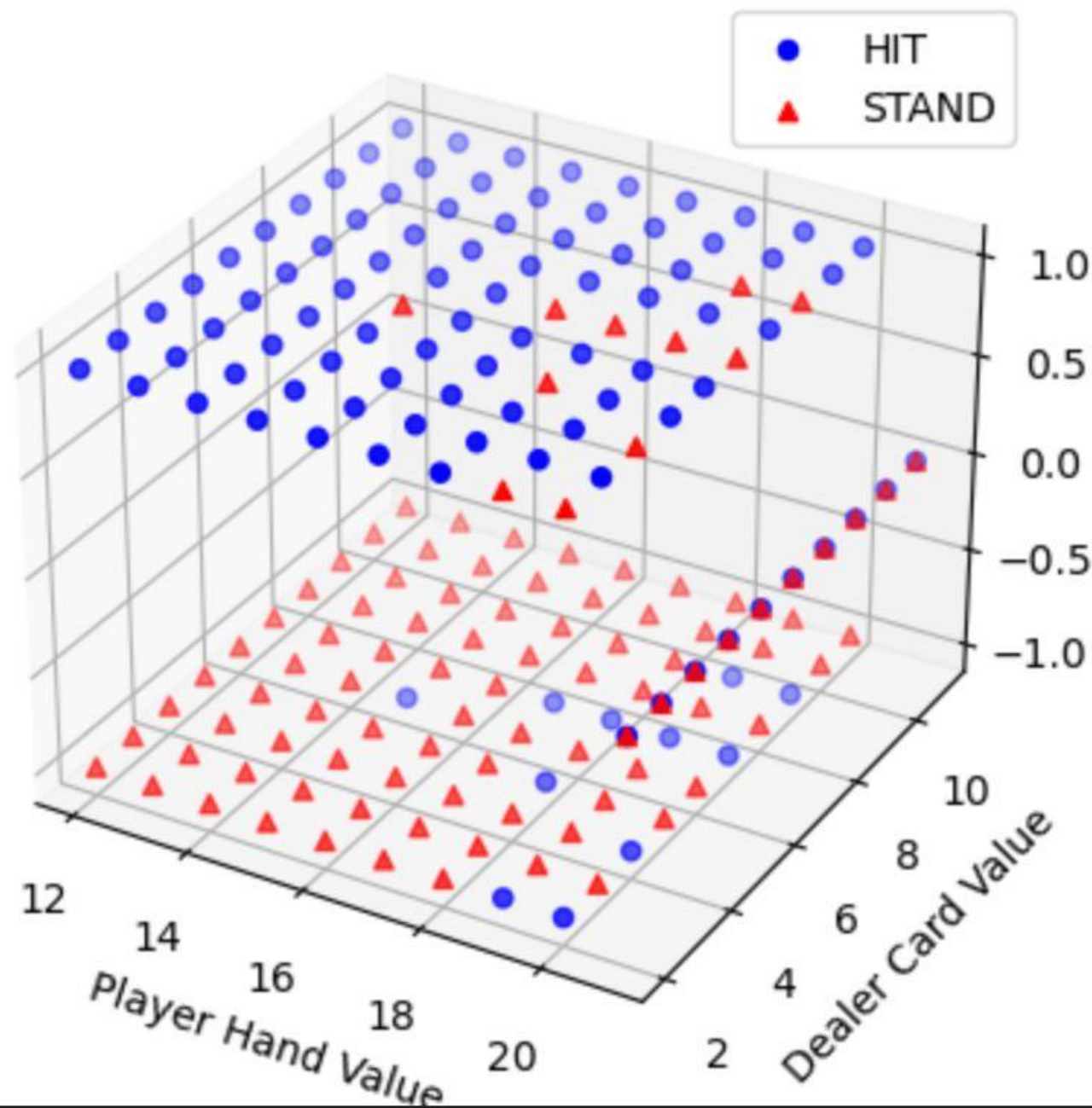
where:

- $Q(s, a)$  is the Q-value for state 's' and action 'a'.
- $\alpha$  (alpha) is the learning rate, determining the step size of the update.
- $\gamma$  (gamma) is the discount factor, balancing immediate and future rewards.
- $s'$  is the next state after taking action 'a' in state 's'.
- $a'$  is the next action chosen according to the behavior policy.

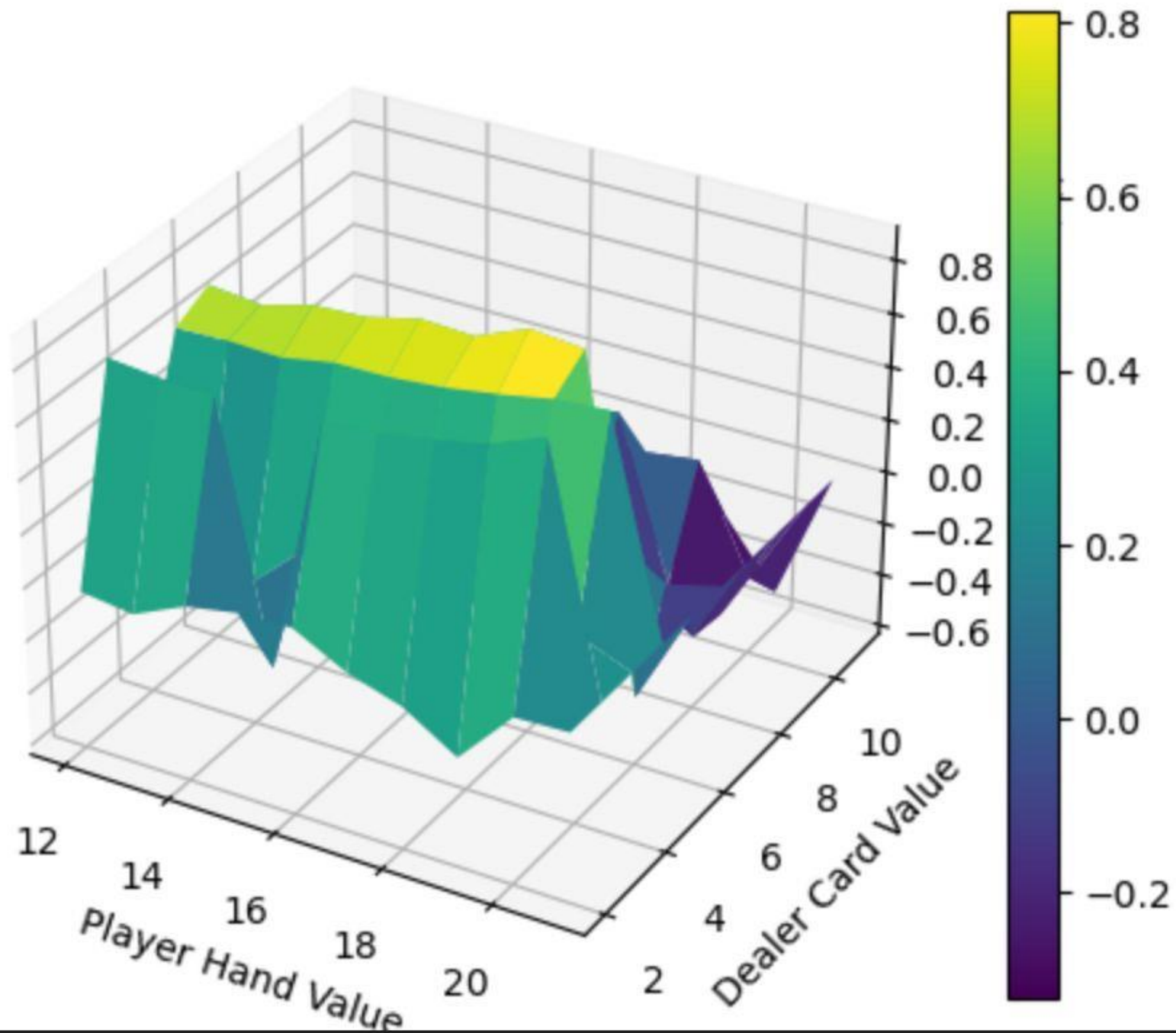


# Learning Curves for Different Exploration Rate Modes in Sarsa









# MONTE- CARLO ALGORITHM

Monte Carlo (MC) algorithm is an on-policy reinforcement learning technique used for learning the optimal policy in Markov Decision Processes (MDPs). Unlike off-policy methods like Q-learning, which learn the value of the optimal policy independently of the agent's behavior policy, Monte Carlo methods learn directly from experience generated by the same policy they are trying to evaluate and improve. This characteristic makes Monte Carlo methods well-suited for scenarios where exploration is desired within the context of the current policy. By collecting complete episodes of experience and averaging returns over multiple episodes, Monte Carlo methods estimate the value of state-action pairs and improve the policy iteratively, ultimately converging to the optimal policy for the given environment.

A red die is positioned on a green game board. In the background, there are yellow chips and cards, including one with the number 9. The scene is brightly lit, suggesting a casino or gaming environment.

# Application into BlackJack - 1

- *Episodic Learning:* In Blackjack, each game can be considered an episode. It starts with the initial deal of cards to the player and the dealer and ends when the game is resolved (either by the player winning, losing, or tying with the dealer). The sequence of state-action-reward tuples observed during the episode consists of the player's current hand, the dealer's visible card, the action taken by the player (hit or stand), and the resulting reward.
- *Policy Evaluation:* MC methods estimate the value of states or state-action pairs directly from experience. In Blackjack, this means estimating the value of each state-action pair, such as (player\_sum, dealer\_card, usable\_ace, action), where player\_sum is the sum of the player's cards, dealer\_card is the dealer's visible card, usable\_ace indicates whether the player has a usable ace, and action is the player's decision to hit or stand. The estimate of a state-action pair's value is the average of the observed returns (cumulative rewards) obtained by taking that action from that state across multiple episodes.



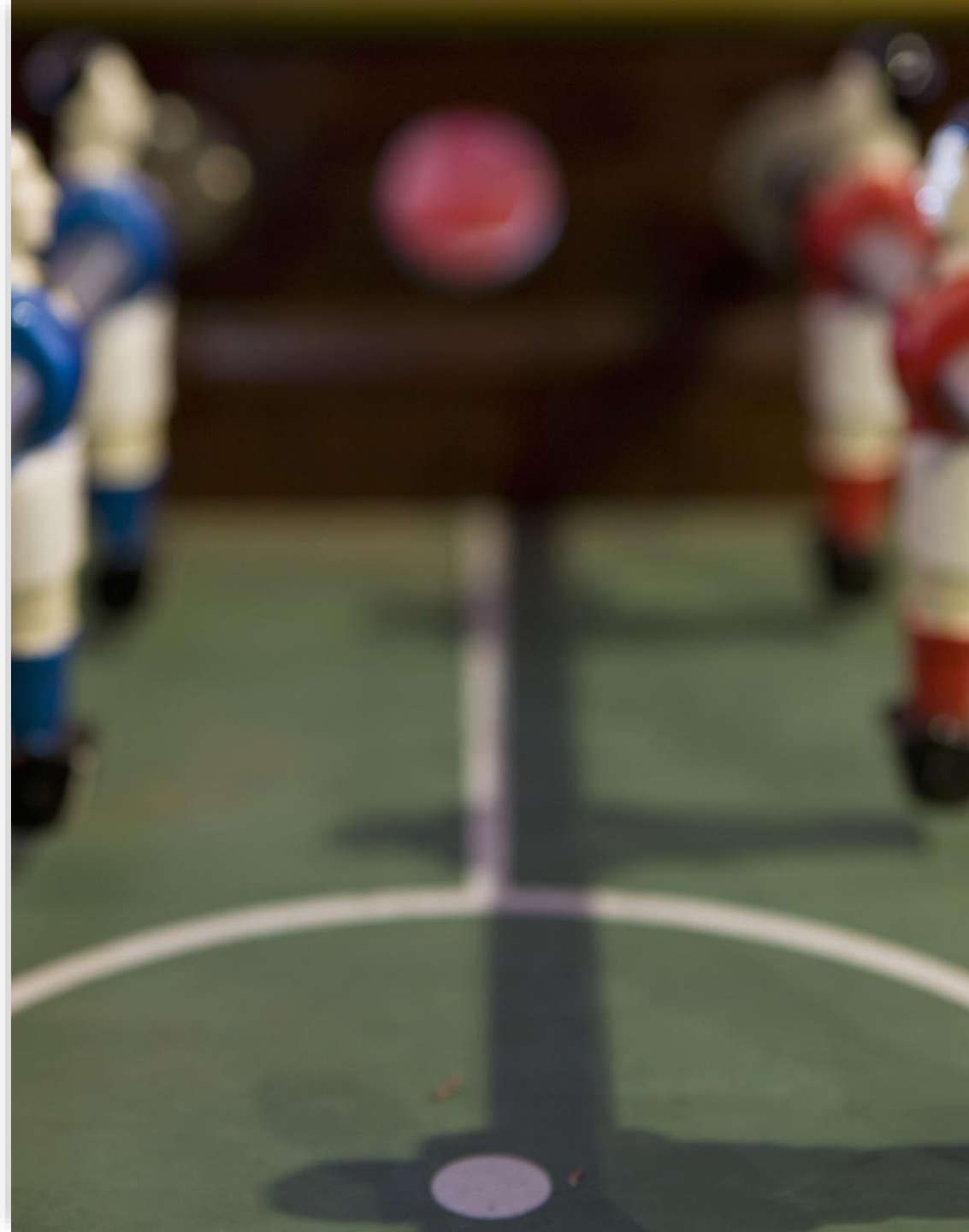
# Application into BlackJack - 2

- *Exploration and Exploitation:* During the game, the MC algorithm explores the state-action space by selecting actions (hit or stand) based on the current policy. In Blackjack, the policy could be an  $\epsilon$ -greedy policy, where the player selects the action with the highest estimated value with probability  $1-\epsilon$  and selects a random action with probability  $\epsilon$  to encourage exploration.
- *Value Estimation:* At the end of each episode (game), the MC algorithm updates the value estimates for the state-action pairs encountered during the episode. For example, if the player wins the game, positive rewards are assigned to each state-action pair leading to the win. These rewards are used to update the corresponding value estimates.



# Application into BlackJack - 3

- *Policy Improvement:* After estimating the value of state-action pairs over multiple episodes, the MC algorithm improves the policy by selecting actions with the highest estimated values. In Blackjack, this means the player will choose to hit if the estimated value of hitting is higher than standing, and vice versa.
- *Training Loop:* The MC algorithm iteratively interacts with the environment by playing multiple episodes of Blackjack. After each episode, the MC algorithm updates the value estimates based on the observed returns and improves the policy accordingly. This process continues until convergence, where the estimated values converge to the true value function and the policy converges to the optimal policy.





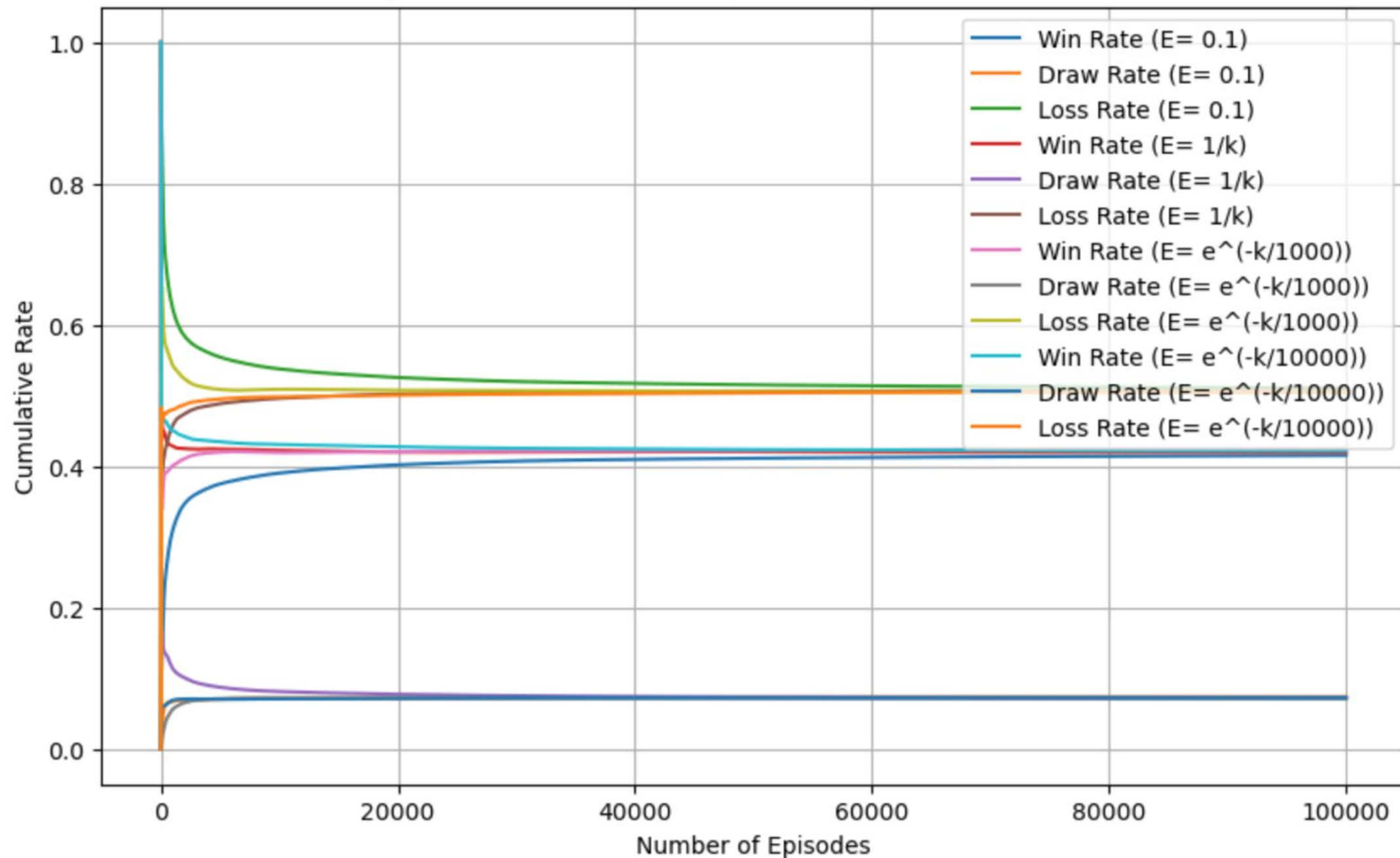
# Formula used in Monte-Carlo Algorithm

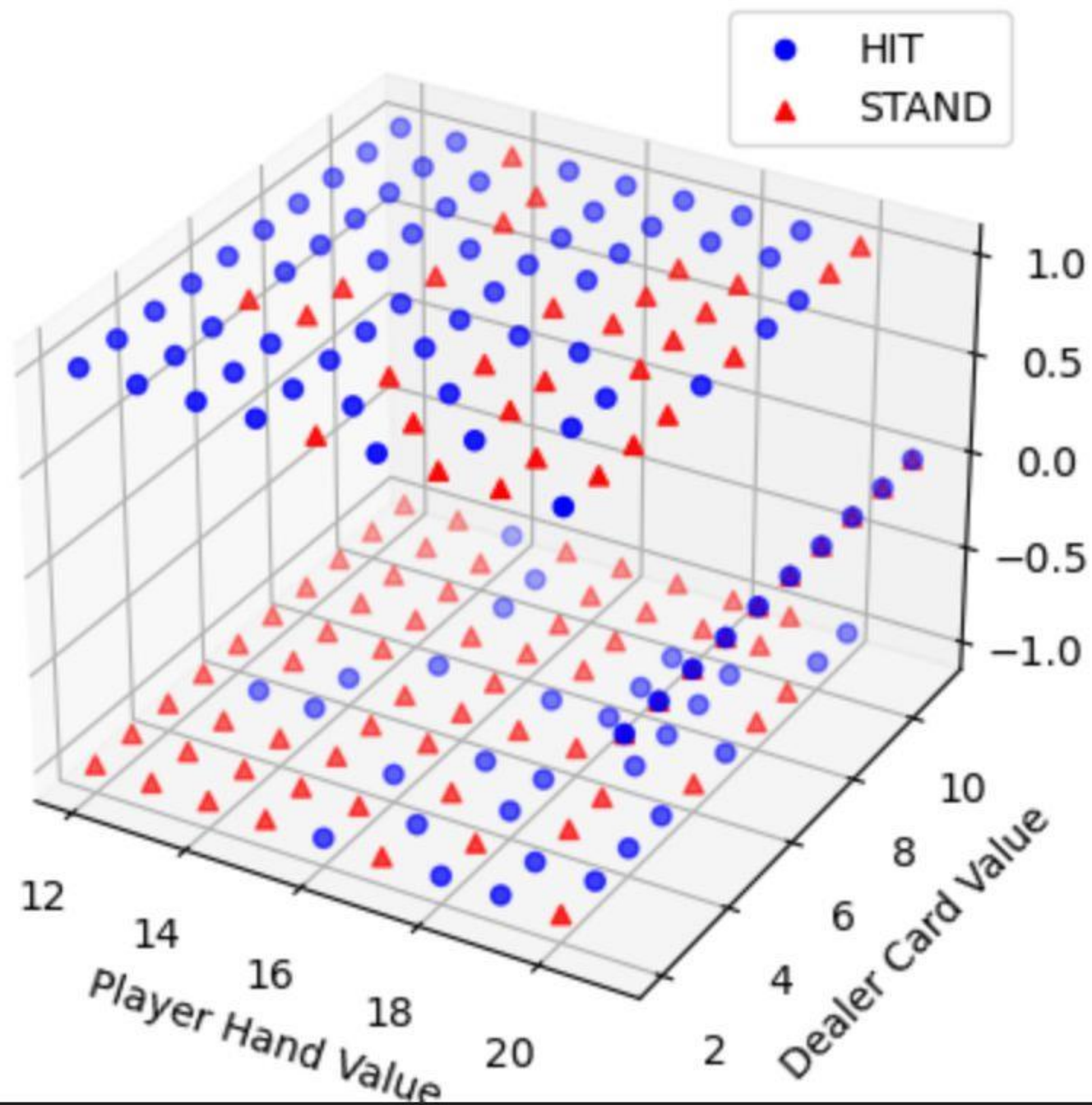
$Q(s, a) = Q(s, a) + \alpha * (G - Q(s, a))$  where:

- $Q(s, a)$  is the current estimate of the action-value function for state  $s$  and action  $a$ .
- $\alpha$  is the learning rate, determining the step size of the update.
- $G$  is the observed return obtained after taking action  $a$  in state  $s$ .
- $s$  represents the current state of the game, typically defined by the player's current sum, the dealer's visible card, and whether the player holds a usable ace.
- $a$  represents the action taken by the player (e.g., hit or stand).
- $G$  is the return obtained after completing the episode, which is the sum of rewards obtained from the initial state until the end of the episode.



# Learning Curves for Different Exploration Rate Modes in MonteCarlo



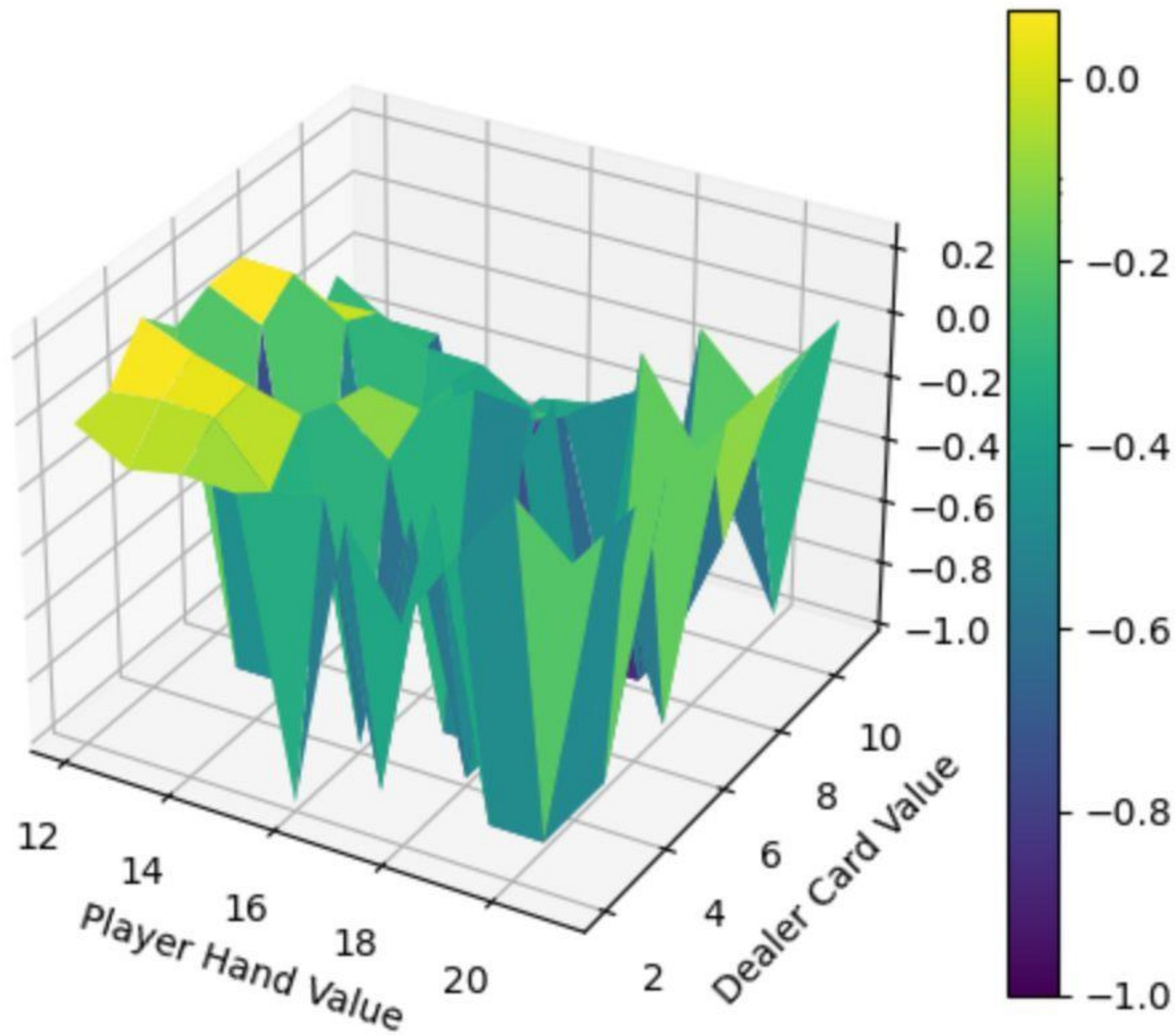


With a usable Ace:

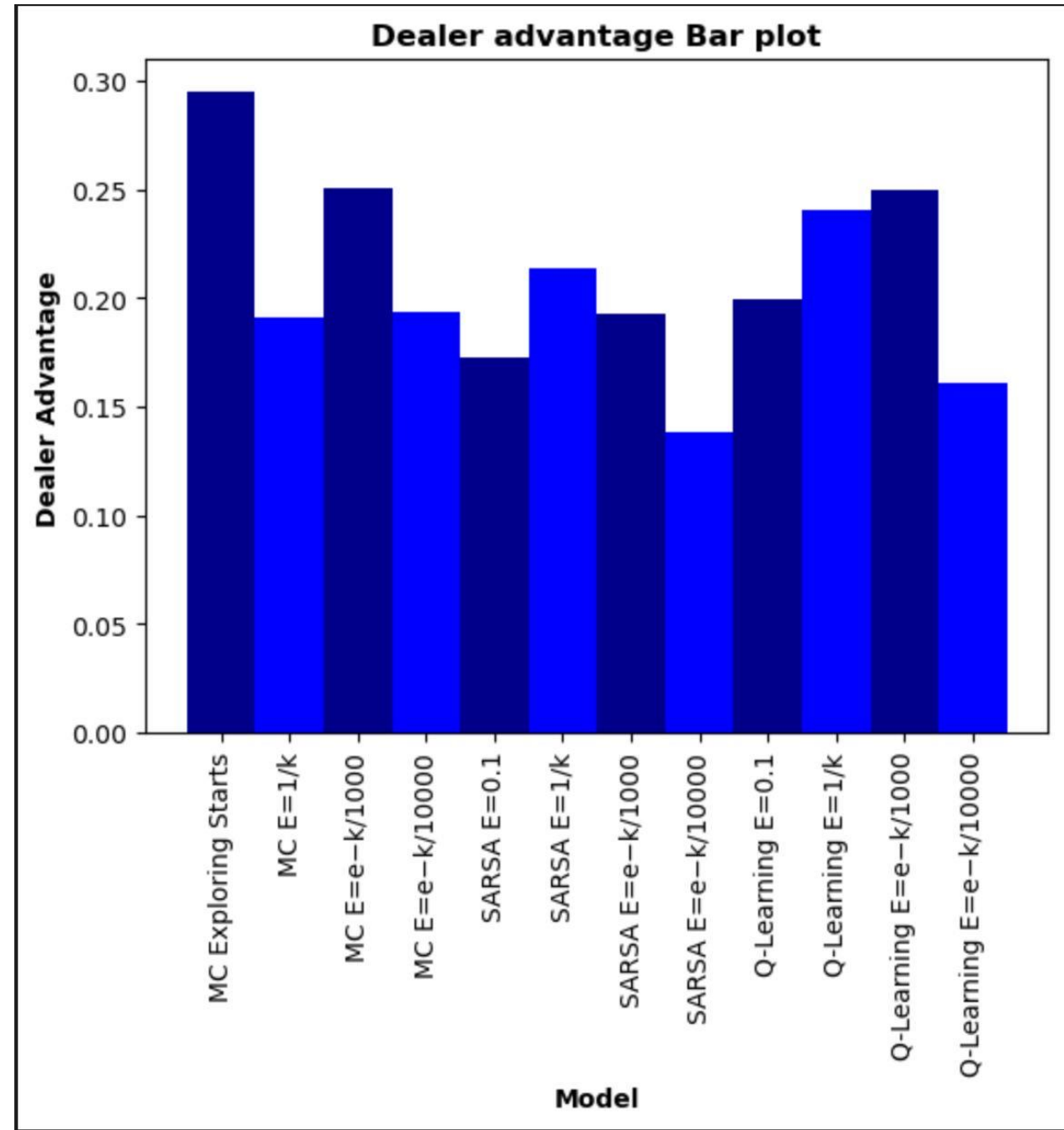
	2	3	4	5	6	7	8	9	10	A
20	H	S	S	S	H	S	H	H	S	S
19	S	S	H	H	S	S	S	S	H	H
18	S	H	S	S	H	S	S	S	H	H
17	H	S	H	S	H	S	H	H	H	H
16	S	H	S	H	H	H	H	H	H	H
15	H	H	H	H	H	S	H	S	S	H
14	H	H	H	S	S	H	H	H	H	S
13	H	H	H	S	H	H	H	H	H	H
12	H	H	H	H	H	H	H	H	H	H

Without a usable Ace:

	2	3	4	5	6	7	8	9	10	A
20	S	S	S	S	S	S	S	H	S	H
19	S	S	S	S	S	S	S	S	H	H
18	S	S	S	S	S	H	H	H	S	H
17	S	H	H	H	S	S	H	S	H	S
16	H	S	H	H	H	H	H	H	H	H
15	S	H	H	H	H	H	H	H	H	H
14	S	H	S	H	H	H	H	H	H	H
13	H	H	S	S	H	H	H	H	H	H
12	S	S	H	H	H	H	H	H	H	S



# Comparison of all algorithms





# **THANK YOU !**

Hope you liked our project !