Blackjack AI: A Q-Learning Approach

# Final Project Report

# 1. Introduction

## Project Overview

This project implements an AI agent that learns to play Blackjack using Q-learning, a model-free reinforcement learning algorithm. The system simulates the game environment, allowing the agent to learn optimal strategies through trial and error. The implementation includes a custom Blackjack environment, Q-learning algorithm, and tools for tracking and visualizing the learning process.

## Problem Statement

Blackjack presents an interesting challenge for AI systems due to its combination of chance and strategy. While basic strategy exists, developing an AI that can learn and adapt its strategy through experience is valuable for both educational and practical purposes. The challenge lies in creating an agent that can learn optimal decisions without explicit programming of game rules or strategies.

## Scope and Challenges

The project scope includes:  
• Implementing a custom Blackjack environment  
• Developing a Q-learning agent  
• Training the agent through simulation  
• Evaluating performance against basic strategy

Key challenges:  
• State space complexity in Blackjack  
• Balancing exploration and exploitation  
• Efficient learning with sparse rewards  
• Measuring and comparing performance

# 2. Background & Related Work

## Existing Approaches

Traditional approaches to Blackjack include:  
• Basic strategy tables  
• Card counting systems  
• Monte Carlo methods  
• Rule-based expert systems

## How This Project Differs

This project differs from existing approaches by:  
• Using Q-learning for strategy development  
• Learning through experience rather than pre-programmed rules  
• Adapting to different game conditions  
• Providing real-time visualization of learning progress

# 4. Methodology

## Baseline Model

The Q-learning implementation uses a tabular approach with the following key components:

class QLearningAgent:  
 def \_\_init\_\_(self, learning\_rate: float = 0.05,   
 discount\_factor: float = 0.95,  
 exploration\_rate: float = 1.0,  
 exploration\_decay: float = 0.9995):  
 self.q\_table = defaultdict(lambda: np.zeros(3)) # 3 actions: hit, stand, double  
 self.lr = learning\_rate  
 self.gamma = discount\_factor  
 self.epsilon = exploration\_rate  
 self.epsilon\_decay = exploration\_decay  
 self.min\_epsilon = 0.005

Key features:  
• Adaptive exploration rate with decay  
• Tabular Q-table for state-action values  
• Three possible actions: hit, stand, double down  
• Learning rate of 0.05 for stable updates  
• Discount factor of 0.95 for future reward consideration

# 5. Technical Implementation

## Environment Design

The Blackjack environment implements casino rules:

class BlackjackEnv:  
 def \_\_init\_\_(self, num\_decks: int = 4):  
 self.deck = Deck(num\_decks)  
 self.player\_hand: List[Card] = []  
 self.dealer\_hand: List[Card] = []  
 self.game\_over = False  
 self.player\_sum = 0  
 self.dealer\_sum = 0  
 self.player\_has\_usable\_ace = False  
 self.dealer\_has\_usable\_ace = False

Features:  
• Multiple deck support  
• Proper ace handling  
• Dealer strategy implementation  
• Natural blackjack detection

# 6. Results

## Performance Metrics

• Training convergence analysis  
• Win rate progression  
• Policy stability measures  
• Computational efficiency

# 7. Conclusions & Future Work

## Key Achievements

• Successful implementation of Q-learning for Blackjack  
• Demonstrated learning capability  
• Efficient state representation  
• Practical training framework

# 9. References

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.  
2. Thorp, E. O. (1966). Beat the Dealer: A Winning Strategy for the Game of Twenty-One. Vintage.  
3. Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.  
4. OpenAI Gym: A toolkit for developing and comparing reinforcement learning algorithms.