# 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

#### 1.1. Development Process & Prompt History

##### Initial Prompt (Claude Sonnet 3.7)

The project began in Claude Sonnet 3.7, where the initial proposal and requirements were uploaded. The goal was to develop a reinforcement learning agent to play blackjack optimally using Python and machine learning.

##### Transition to Cursor IDE

Due to export and constraint issues in Claude, a summary of the chat was requested to continue the project in Cursor IDE. The summary included the project's structure, missing components, and the course context (CIS 730).

##### Iterative Prompting and Development in Cursor

Prompts were used to:

* Add a deterministic baseline agent (basic strategy)
* Add a random agent for baseline comparison
* Implement statistical analysis (binomial test, p-value calculation)
* Compare Q-learning, basic strategy, and random agents
* Fix environment logic to prevent infinite loops
* Update the README and documentation for clarity and reproducibility
* Push all changes to GitHub for version control

Example Prompts:

* "Add a deterministic policy (hardwired table) for comparison."
* "Add a random agent for baseline comparison."
* "Implement p-value calculation for a series of iid Bernoulli random variables."
* "Fix the environment so that games always terminate."
* "Update the README to include agent comparison and statistical analysis."
* "Commit and push all changes to GitHub."

### 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

### 3. Dataset and Processing

#### Data Sources

The project uses simulated game data generated through:

* Custom Blackjack environment
* Random game play for initial exploration
* Policy-guided play during training

#### Preprocessing and Filtering

* State representation: (player\_sum, dealer\_card, usable\_ace)
* Action space: {hit, stand, double}
* Reward structure: {-2, -1, 0, 1} for different outcomes
* Normalization of state values

### 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

#### State Representation

The environment uses a compact state representation:

def \_get\_state(self):  
 """Return the current state of the game"""  
 dealer\_showing = self.dealer\_hand[0].value  
 return (self.player\_sum, dealer\_showing, int(self.player\_has\_usable\_ace))

State components:

1. Player's current sum (4-21)
2. Dealer's showing card (1-10)
3. Whether player has a usable ace (0 or 1)

#### Training Process

The training loop implements the Q-learning update rule:

def update(self, state, action, reward, next\_state, done):  
 best\_next\_action = np.argmax(self.q\_table[next\_state])  
 td\_target = reward if done else reward + self.gamma \* self.q\_table[next\_state][best\_next\_action]  
 td\_error = td\_target - self.q\_table[state][action]  
 self.q\_table[state][action] += self.lr \* td\_error

Key aspects:

* Temporal Difference (TD) learning
* Epsilon-greedy exploration
* Adaptive learning rate
* Experience replay buffer

### 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

#### Action Space

The environment supports three actions:

def step(self, action):  
 if action == 0: # Stick  
 return self.\_dealer\_play()  
 elif action == 1: # Hit  
 self.player\_hand.append(self.deck.deal())  
 self.\_calculate\_hand\_value()  
 elif action == 2: # Double down  
 if len(self.player\_hand) > 2:  
 return self.\_get\_state(), -1, False, info

#### Reward Structure

The reward system is designed to encourage optimal play:

def \_dealer\_play(self):  
 if self.dealer\_sum > 21:  
 return self.\_get\_state(), 1, True, info # Win  
 elif self.dealer\_sum > self.player\_sum:  
 return self.\_get\_state(), -1, True, info # Loss  
 elif self.dealer\_sum < self.player\_sum:  
 return self.\_get\_state(), 1, True, info # Win  
 else:  
 return self.\_get\_state(), 0, True, info # Draw

Rewards:

* +1 for winning
* -1 for losing
* 0 for drawing
* -2 for busting after double down

#### Performance Optimization

Key optimizations include:

1. Efficient State Representation:

# Using tuples for immutable state representation  
state = (player\_sum, dealer\_showing, int(has\_usable\_ace))

1. Memory-Efficient Q-Table:

# Using defaultdict for sparse state representation  
self.q\_table = defaultdict(lambda: np.zeros(3))

1. Batch Processing:

# Vectorized operations for Q-updates  
td\_error = td\_target - self.q\_table[state][action]  
self.q\_table[state][action] += self.lr \* td\_error

### 6. Results (Updated)

#### Agent Comparison & Statistical Analysis

This project compares three agents:

* \*\*Q-Learning Agent:\*\* Learns optimal play through reinforcement learning.
* \*\*Basic Strategy Agent:\*\* Plays using a deterministic, hardcoded basic strategy table.
* \*\*Random Agent:\*\* Chooses actions randomly.

##### Statistical Evaluation

After running a series of games, the win rates of each agent are compared.

A \*\*binomial test\*\* is used to calculate the p-value for each agent's win rate against the theoretical win rate for perfect basic strategy (42%).

\*\*Null Hypothesis (H₀):\*\*

There is no significant difference in win rate between the Q-learning agent and the basic strategy(42%) or random agent.

\*\*Alternative Hypothesis (H₁):\*\*

The Q-learning agent has a significantly higher win rate than the basic strategy or random agent.

\*\*P-value Interpretation:\*\*

* A low p-value (< 0.05) means the agent's win rate is significantly different from 42%.
* A high p-value (≥ 0.05) means the agent's win rate is not significantly different from 42%.

##### Example Output

Q-Learning Agent:  
 Wins: 377 (37.7%)  
 Losses: 497  
 Pushes: 126  
 p-value vs 42% win rate: 0.0059  
  
Basic Strategy Agent:  
 Wins: 390 (39.0%)  
 Losses: 482  
 Pushes: 128  
 p-value vs 42% win rate: 0.0546  
  
Random Agent:  
 Wins: 239 (23.9%)  
 Losses: 684  
 Pushes: 77  
 p-value vs 42% win rate: 0.0000

### 7. Conclusions & Future Work

#### Key Achievements

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

#### Future Improvements

* Deep Q-learning implementation
* Multi-agent training
* Real-time visualization
* Web interface development

### 8. Challenges Encountered

#### Development Challenges

* State space complexity
* Exploration-exploitation balance
* Training stability
* Performance optimization

#### Solutions

* Adaptive exploration strategy
* Experience replay implementation
* Hyperparameter tuning
* Efficient data structures

### 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.