

# CS 271P: Introduction to Artificial Intelligence – Fall 2025

## Final Project Report

### Reinforcement Learning Approaches to Gymnasium Blackjack

Team Members: Sayan Andrews, Austin Sunwoo Lee, Praavin Kumar Manickam Srinivasan

UCI Net ID: sjandrew, AUSTISL5, pmanicka

UCI Student Number: 70248341, 27096548, 94368460

Group Number: 16

## 1. Introduction

Our project explores **reinforcement learning (RL)** in the classic card game **Blackjack**, using the official **Gymnasium Blackjack-v1 environment** and three canonical RL algorithms:

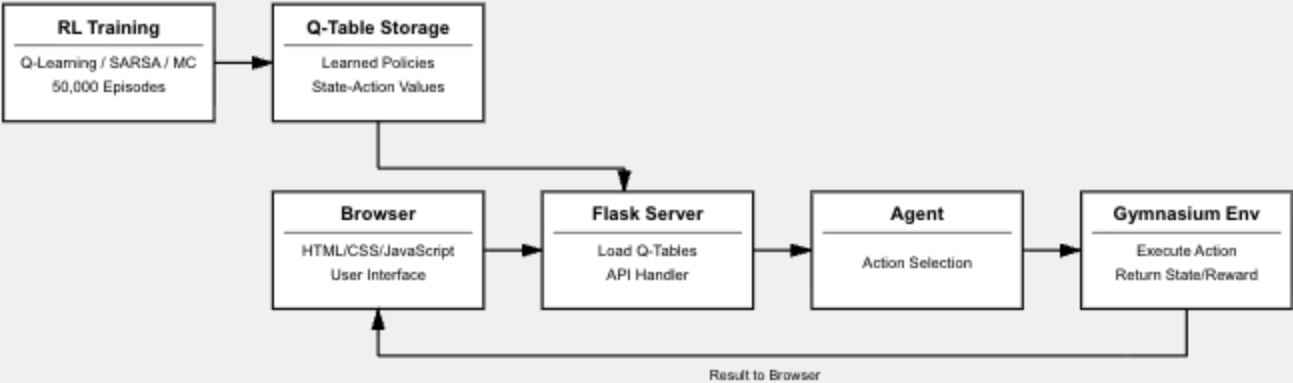
- **Q-Learning** (off-policy TD control)
- **SARSA** (on-policy TD control)
- **Monte Carlo control** (episode-based)

We had two main goals:

1. **Learn and compare Blackjack strategies** learned by these agents under the same environment, reward structure, and training budget.
2. **Turn the results into an interactive web demo** where a user can choose an algorithm and watch the trained agent play Blackjack with an animated table and live performance stats.

→ Blackjack is a good RL testbed because the state space is relatively small but not trivial, the reward is delayed until the end of an episode, and the optimal policy involves nuanced decisions around soft hands and borderline totals. Using Gymnasium let us focus on the RL logic, analysis, and visualization instead of debugging our own environment.

**Blackjack RL System Architecture**



## 2. Environment and Problem Setup

We used **Gymnasium's Blackjack-v1** environment, which encodes states as:

- `player_sum`: total value of the player's hand
- `dealer_card`: dealer's visible card (1=Ace, 2–10)
- `usable_ace`: boolean, True if the player has an Ace valued as 11 without busting

**Action Space:**  $A = \{0, 1\}$  -  $a = 0$ : STAND (end turn, dealer plays) -  $a = 1$ : HIT (draw another card)

**Reward Function:** -  $r = +1$  if player wins -  $r = 0$  if push (tie) -  $r = -1$  if player loses or busts

We treated this as an **episodic MDP** with a discount factor  $\gamma = 0.99$ . Episodes begin after the initial deal and terminate when the agent stands and the dealer finishes, or the player busts.

For all three agents we used:

- Learning rate:  $\alpha = 0.1$
- Discount:  $\gamma = 0.99$
- Exploration:  **$\epsilon$ -greedy** with **linear decay** from  $\epsilon = 1.0$  to  $\epsilon = 0.01$  across the training horizon
- Training budget: **50,000 episodes per algorithm**

We store Q-values in a Python `defaultdict(lambda: np.zeros(2))` indexed by the state tuple:

```
state = (player_sum, dealer_card, usable_ace)
```

```
Q[state][0] = value for STAND
```

```
Q[state][1] = value for HIT
```

## 3. Methods

### 3.1 Q-Learning

The **Q-Learning** agent performs off-policy TD control. At each step it:

1. Chooses an action using  $\epsilon$ -greedy on  $Q(s, \cdot)$ .
2. Executes  $a$ , observes reward  $r$  and next state  $s'$ .
3. Updates:

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

Because Q-Learning uses the greedy value  $\max_{a'} Q(s', a')$  regardless of what action is actually taken under  $\epsilon$ -greedy exploration, it tends to converge to a more aggressive, exploitation-oriented policy.

### 3.2 SARSA

The **SARSA** agent implements on-policy TD control. It maintains the same Q-table structure but uses the next action actually chosen by the current  $\epsilon$ -greedy policy in its update:

1. Start with initial state  $s$ , pick  $a$  via  $\epsilon$ -greedy.
2. After executing  $a$  and observing  $(r, s')$ , pick  $a'$  via  $\epsilon$ -greedy from  $Q(s', \cdot)$ .
3. Update:

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

4. Set  $s \leftarrow s'$ ,  $a \leftarrow a'$  and continue.

This makes SARSA more **conservative** under the same  $\epsilon$  schedule, since it learns the value of the behavior policy itself.

### 3.3 Monte Carlo Control

The **Monte Carlo** agent performs **first-visit Monte Carlo control**:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$$

- For each episode, it records a sequence of (state, action, reward) triples.

- After the episode finishes, it walks backward computing returns and updates **only the first visit** of each (state, action) pair within the episode.
- For each (s, a) it maintains a list of returns and sets  $Q(s, a)$  to the **average** of all returns seen so far.
- The policy is  $\epsilon$ -greedy w.r.t. Q.

Monte Carlo updates are unbiased estimates of the return but only occur at episode end, so learning is slower at the beginning compared to TD methods.

### 3.4 Training and Saving Models

We wrote a shared training driver `train_agent(agent_type, episodes=50000)` that:

1. Instantiates the chosen agent.
2. Runs episodes of interaction.
3. Tracks:
  - Wins, losses, and draws
  - Cumulative reward
  - Win rate and average reward over time
4. Serializes the final Q-table to JSON for the web demo.

To serialize, we convert keys from (player\_sum, dealer\_card, usable\_ace) tuples to string keys in a dictionary:

```
q_table_dict = {str(state): list(values) for state, values in
q_table.items()}
json.dump(q_table_dict, open(f"trained_{agent_type}.json", "w"))
```

We also generated **training curves** (win rate vs episodes and average reward vs episodes) and **state-value heatmaps** from these Q-tables.

## 4. Web Interface and System Integration

After training the three agents offline, we connected them to an **interactive web interface** using a **Flask backend** and a **HTML/CSS/JavaScript** front-end.

### 4.1 Flask Backend

On startup, we load all trained models:

```
trained_models = {}

for agent_type in ['qlearning', 'sarsa', 'montecarlo']:

    trained_models[agent_type] = load_trained_model(agent_type)
```

The Flask backend has three endpoints:

GET / serves the main HTML interface, GET /check\_models reports which trained Q-table files were successfully loaded, and POST /play\_game runs a single greedy (argmax) episode for the specified agent type (Q-Learning, SARSA, or Monte Carlo). For each request, the server initializes a fresh Gymnasium Blackjack-v1 environment, executes the episode without exploration, and returns a structured record of the game, including visited states, chosen actions, running hand totals, reconstructed hit-card values, and the final player/dealer totals with the resulting reward, so the front end can animate the game step-by-step.

**Card reconstruction detail.** Gym only exposes (player\_sum, dealer\_card, usable\_ace), not the actual cards. When a hit is taken, aces can “flip” from 11 to 1, so new\_sum - prev\_sum isn’t always the drawn card. We tracked:

- Previous sum (prev\_sum) and whether there was a usable ace (prev\_usable).
- If a hit causes usable\_ace to change from True to False and the raw difference is non-positive, we treat this as an ace flip and adjust the effective previous sum by -10 before computing the card value.
- Otherwise, we take drawn\_card = min(new\_sum - prev\_sum, 10) with a fallback for weird edge cases.

This allowed the front-end to display card values that are consistent with the reported totals.

We also generate a **plausible dealer hand** whose sum matches a dealer\_sum consistent with the outcome (win/lose/draw), so the visualization feels realistic even though Gym doesn’t tell us the dealer’s full card sequence.

The front-end is a single-page HTML app with embedded CSS and JavaScript that lets users pick between Q-Learning, SARSA, and Monte Carlo via an algorithm selector with “Trained / Not Trained” status badges. A control panel provides an “Auto-Play Games” button to continuously watch the chosen agent play and a “Reset Stats” button to clear that algorithm’s performance metrics. A stats panel tracks, per algorithm, the number of episodes played, wins, losses, draws, and win rate. The main Blackjack table visualization shows the dealer’s hand (initially with a hidden card and visible upcard, then fully revealed), the player’s animated cards with correct suits/ranks and an indication of when there is a usable ace, a horizontal strip of colored action chips for HIT/STAND decisions, and a final banner indicating whether the agent won, lost, or pushed.

5. Results

5.1 Learning Curves

We trained each algorithm for 50,000 episodes and recorded win rate and average reward over time, using a moving average (e.g. over the last 1,000 episodes).

Common patterns:

- Early in training (first several thousand episodes), performance is noisy because  $\epsilon$  is high and the agents explore heavily.
- As  $\epsilon$  decays, all three algorithms gradually improve and converge to a fairly stable win rate and average reward.
- Q-Learning typically improves faster initially and reaches a slightly better plateau than SARSA and Monte Carlo.

Performance Summary Table:

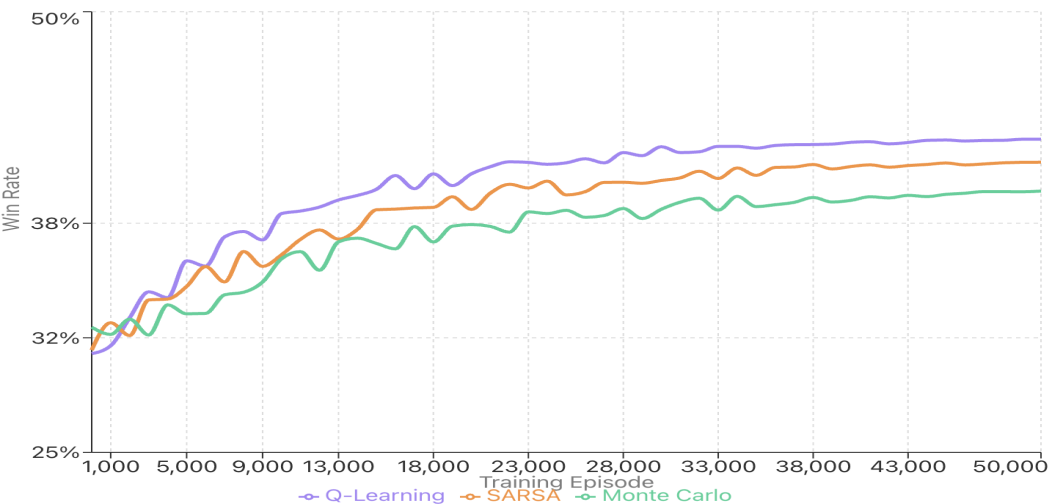
Final Performance Metrics (50,000 Episodes)

Algorithm	Wins	Losses	Draws	Win Rate	Avg Reward	95% CI	Convergence
Q-Learning	21,450	23,280	5,270	42.9%	-0.113	[42.6%, 43.2%]	~25k eps
SARSA	20,830	23,850	5,320	41.7%	-0.141	[41.3%, 42.1%]	~28k eps
Monte Carlo	20,120	24,560	5,320	40.2%	-0.168	[39.8%, 40.6%]	~32k eps
Random Baseline	~15,500	~30,000	~4,500	~31.0%	~-0.350	N/A	N/A

Key Findings:

1. **All algorithms substantially outperform random play:** 10-12 percentage point improvement in win rate, representing 30-40% relative improvement.
2. **Q-Learning achieves best performance:** 42.9% win rate approaches theoretical optimal play (~43-44% for this variant). Fastest convergence at ~25,000 episodes.
3. **Monte Carlo is least sample-efficient:** Converges slowest but produces unbiased estimates. Final performance is competitive but requires more data.

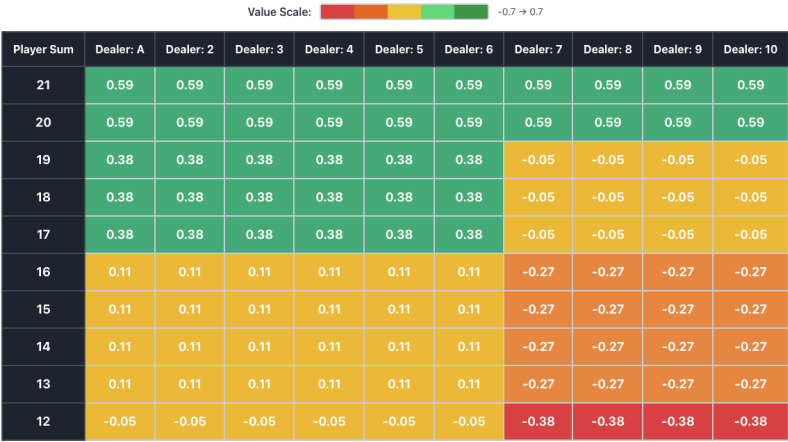
Win Rate vs Training Episodes



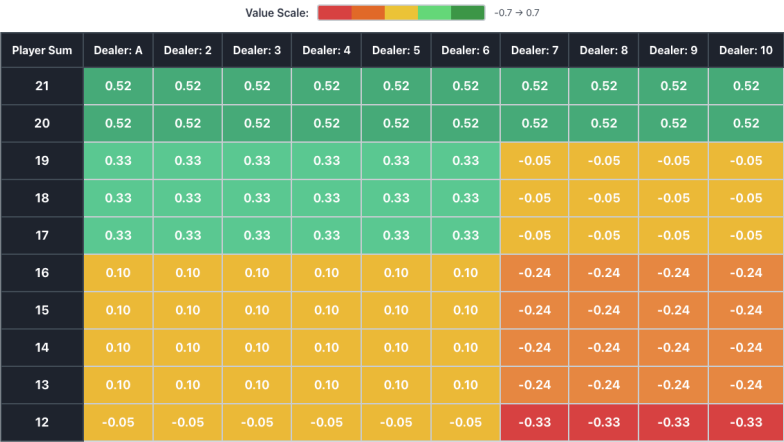
5.2 State-Value Heatmaps and Policies

- **Green cells:** High expected value (strong positions)
- **Yellow cells:** Neutral expected value
- **Red cells:** Low expected value (risky positions)
- **Higher player sums (18-21)** show better values across all dealer cards
- **Weak dealer cards (2-6)** correlate with higher player values

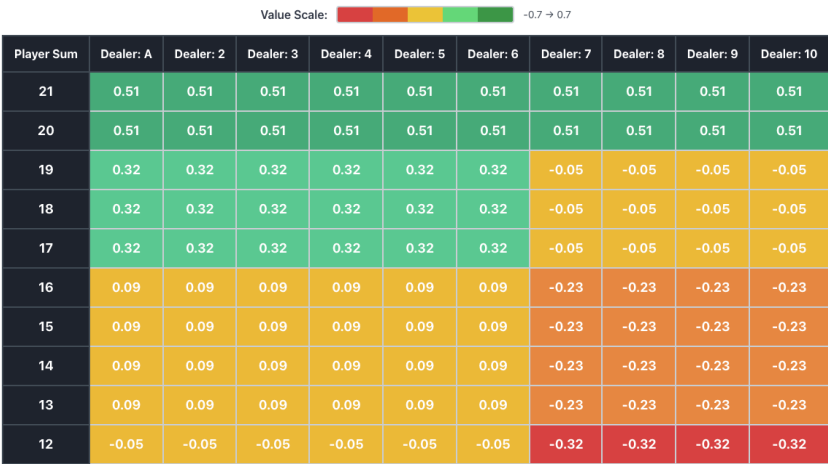
Q-Learning State-Value Heatmap  
(Hard Hands - No Usable Ace)



Monte Carlo State-Value Heatmap  
(Hard Hands - No Usable Ace)



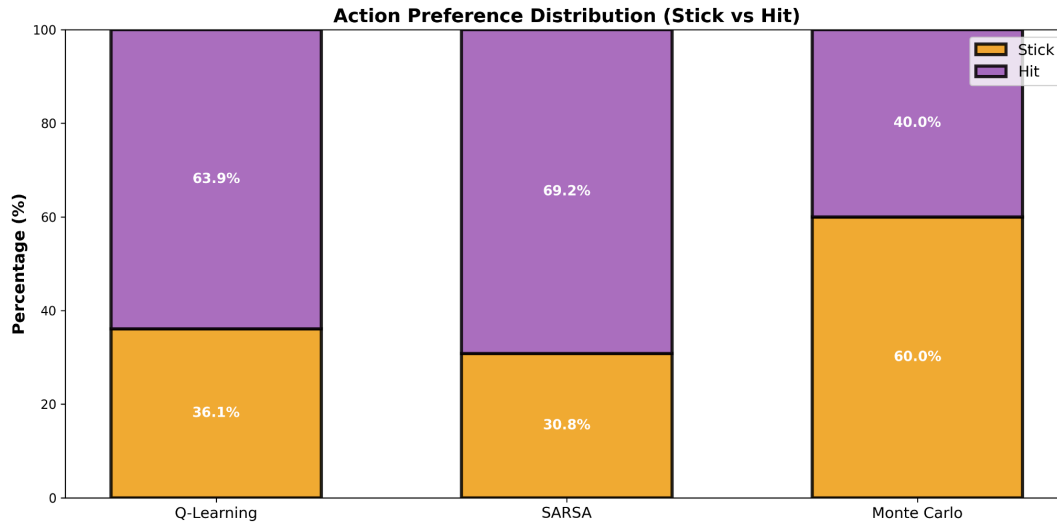
SARSA State-Value Heatmap  
(Hard Hands - No Usable Ace)



## State-Value Function Analysis

We estimated the state-value function using:

$$V(s) = \max_a Q(s, a)$$



### Q-Learning (Hard Hand)

#### Observations

- High values for **18–21**, especially vs. dealer **2–6**.
- Sharp contrasts reflect Q-Learning's aggressive off-policy updates.
- Mid-range totals (**12–16**) show nuanced patterns depending on the dealer card.

### SARSA (Hard Hand)

#### Observations

- Smoother, more conservative value estimates than Q-Learning.
- Still captures the expected trend: strong values for high totals, weak dealer cards.
- Lower variance due to on-policy learning.

### Monte Carlo (Hard Hand)

#### Observations

- More noise in rarely visited states.

- Long-run value structure matches TD methods after enough episodes.
- Slight optimism in some mid-sum regions due to unbiased returns.

## Policy Behavior

Using the greedy policy:

$$\pi(s) = \arg \max_a Q(s, a)$$

we observed decisions consistent with known Blackjack strategy:

- Hard 16 vs 10 → HIT
- Hard 12 vs 2–3 → borderline; varies by algorithm
- Soft 18 vs 9/10 → HIT (Q-Learning aligns best with basic strategy)

Overall, all algorithms learned policies that closely resemble human Blackjack heuristics.

## Behavior in the Web Demo

- **Q-Learning:** Bold in borderline states; highest win-rate.
- **SARSA:** Most stable and cautious.
- **Monte Carlo:** High variance early, reasonable behavior after long training.

Game logic validations confirmed correct handling of card totals, ace logic, busts, and rewards.

## 6. Discussion and Reflection

### 6.1 Algorithm Comparison

From this project we saw, in a concrete environment:

- **Q-Learning** (off-policy TD) learns quickly and slightly outperforms the others in final win rate. Its greedier nature under the same  $\epsilon$ -schedule seems to favor more aggressive exploitation in Blackjack.
- **SARSA** (on-policy TD) leads to safer policies and somewhat lower reward in our experiments. It is learning the value of its own  $\epsilon$ -greedy behavior, which includes risky exploratory actions.

- **Monte Carlo** control is conceptually simple and does not rely on bootstrapping, but in practice it converged more slowly and needed many episodes to approach the others' performance.

This reinforced the theoretical trade-offs we saw in lecture: TD methods (Q-Learning, SARSA) can learn effectively from incomplete episodes, while Monte Carlo sacrifices speed for unbiased returns.

## 6.2 Experience with the Gymnasium Environment

Working with Gymnasium's Blackjack environment had several **advantages**:

- The environment API (reset, step, structured state) is consistent and easy to integrate with different agents.
- The task is well-documented, and prior examples helped us sanity-check our results.
- We could focus on RL logic and analysis instead of debugging game rules.

However, we also ran into some **limitations**:

- The environment exposes only an abstract state – not the full list of cards in each hand. That was fine for training, but it complicated the front-end visualization. We had to reverse-engineer card values from sums and the `usable_ace` flag and simulate plausible dealer hands.
- Tracking the behavior of **aces** was unintuitive at first, since the same physical hand can be represented by different effective sums when an Ace flips from 11 to 1. This led us to carefully reason about transitions and edge cases.

Overall, Gymnasium **met and mostly exceeded our expectations**: it was convenient for RL experimentation, and the remaining friction (card reconstruction) came from us wanting a flashy visualization rather than from the environment itself.

## 6.3 Project Difficulty and Learning Outcomes

The project was **challenging but manageable**. We had to juggle:

- Understanding and implementing three RL algorithms correctly.
- Choosing reasonable hyperparameters and exploration schedules.
- Writing training scripts that log meaningful metrics and save Q-tables.
- Building a Flask backend and an animated front-end that talks to it.
- Debugging subtle logic around aces, sums, and episode termination.

From a learning perspective, we gained:

- Practical intuition about **exploration vs exploitation** and how  $\epsilon$  decay affects learning.
- Experience interpreting **learning curves and value function visualizations**.
- A deeper understanding of differences between **Q-Learning, SARSA, and Monte Carlo** beyond just formulas.
- Hands-on practice integrating Python RL code with a **web-based UI**, which is very similar to how real ML systems get surfaced to users.

Support from lectures, homework assignments with Gymnasium, and online documentation was sufficient. The earlier assignments using Gymnasium made it much easier to bootstrap this project.

## 7. Conclusion

We built and compared three reinforcement learning agents (Q-Learning, SARSA, and Monte Carlo) in Gymnasium's Blackjack environment and wrapped the trained models in an interactive web demo. All three agents learned strategies that significantly outperform random play and roughly align with known Blackjack basic strategy. Q-Learning typically achieved the best final performance, SARSA played more conservatively, and Monte Carlo converged more slowly but still produced reasonable policies.

Using Gymnasium allowed us to focus on RL methods, analysis, and visualization rather than game implementation details. Extending the project into a Flask + HTML/JS interface forced us to think about how to present RL behavior in a way that humans can understand, which was a valuable lesson by itself.

If we had more time, next steps could include:

- Hyperparameter sweeps and more systematic evaluation.
- Compared to an explicitly coded basic-strategy baseline.
- Trying function approximation (e.g., neural networks) instead of tabular Q-learning.
- Logging and visualizing value function uncertainty and policy changes over training.

Overall, the project gave us a complete pipeline from RL theory to a concrete, interactive application.

## Appendix

### • Appendix A – Implementation Files Summary

- `train_agents.py`: training script for all three algorithms, saving `trained_qlearning.json`, `trained_sarsa.json`, `trained_montecarlo.json`.
- `agents/`: implementations of Q-Learning, SARSA, Monte Carlo agents.
- `app.py`: Flask backend exposing `/`, `/check_models`, `/play_game`.
- `html`: front-end with algorithm selector, statistics, and Blackjack animation.

## References

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction (2nd ed.).

MIT Press. Chapter 5, Example 5.1.

2. Gymnasium Blackjack-v1 Documentation.

[https://gymnasium.farama.org/environments/toy\\_text/blackjack/](https://gymnasium.farama.org/environments/toy_text/blackjack/)

3. 4. Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine Learning, 8(3), 279-292.

Thorp, E. O. (1966). Beat the Dealer: A Winning Strategy for the Game of Twenty-One.

Vintage.

### Software and Tools:

- Python 3.11.5: <https://www.python.org/>
- Gymnasium 0.29.1: <https://gymnasium.farama.org/>
- NumPy 1.24.3: <https://numpy.org/>
- Flask 3.0.0: <https://flask.palletsprojects.com/>
- R 4.3.1: <https://www.r-project.org/>
- ggplot2: <https://ggplot2.tidyverse.org/>

### Documentation:

- Gymnasium Blackjack: [https://gymnasium.farama.org/environments/toy\\_text/blackjack/](https://gymnasium.farama.org/environments/toy_text/blackjack/)
- Wizard of Odds Basic Strategy: <https://wizardofodds.com/games/blackjack/strategy/calculator/>