CS 329 - FOUNDATIONS OF AI: MULTIAGENT SYSTEMS

Gridworld Reinforcement Learning

Fall 2025

Submitted by:

Revathi Katta (23110159)
Pappala Sai Keerthana (23110229)
Ravi Bhavana (23110274)
Thoutam Dhruthika (23110340)

Explore our interactive Gridworld environment here:

gridworld-rl-agent

Contents

1 Introduction

Project Title: Gridworld Reinforcement Learning

Reinforcement Learning (RL) is a framework in which an agent learns to make sequential decisions by interacting with an environment and receiving rewards. Unlike supervised learning, where the correct labels are known, RL agents must discover optimal actions through trial and error.

This project implements RL algorithms within a classic Gridworld environment. The primary aim is to compare the behaviour of two foundational RL methods—Q-Learning and SARSA—on the same environment to study policy stability, convergence differences, and behavioural tendencies under identical reward structures.

The environment used is a configurable Gridworld supporting grid sizes from 3×3 up to 15×15. The agent can move in four directions, must avoid holes, and aims to reach a high-reward goal state. The experiments help visualise and understand how different RL algorithms behave under identical conditions, particularly how on-policy and off-policy learning differ in practice.

2 Problem Definition

Gridworld is a discrete environment structured as a 2D grid. The agent begins at the start cell (state 0) and aims to reach the goal, located at the final cell of the grid (state $N-1$, where N is the total number of cells).

The objective of the agent is:

- Reach the terminal goal state to obtain a high reward.
- Avoid hole states, which act as negative terminal states.
- Maximise cumulative reward over the episode.

The interaction is episodic: each episode begins at the start state and ends when the agent reaches either a goal or a hole. No stochasticity is built into the environment transitions—movement is deterministic. The only source of randomness is the agent's exploration policy, implemented using the standard ϵ-greedy method. With probability $1-\epsilon$, the agent chooses the best known action; with probability ϵ, it selects a random action to explore.

3 Environment Design

The environment is a standard **Gridworld** designed for comparative Reinforcement Learning studies.

3.1 Gridworld Specification

- **Layout**: Dynamic grid size (e.g., 3×3 to 15×15), with 4×4 as the default.

- **States**:
  - **Start**: Always the first cell (State 0).
  - **Goal**: The last cell of the grid (High positive reward, episode ends).
  - **Holes**: Negative absorption states (Penalty, episode ends).
- **Actions**: Four deterministic movements: Up, Down, Left, Right.
- **Boundaries**: Attempting an out-of-bounds move results in the agent remaining in the current cell.

3.2 Reward Structure and Terminal Dynamics

The reward function incentivizes **efficient navigation** while **avoiding risk**. The environment transitions are **deterministic** and managed by the `step()` function, which returns the next state, the resulting reward, and the `done` flag.

Table 1: Gridworld Reward Function $\mathcal{R}(s, a, s')$

Event	**Reward**	**Episode Status**
Reaching the **Goal** state	$+1000$	Terminates (`done` = `True`)
Falling into a **Hole** state	-100	Terminates (`done` = `True`)
Each **Non-Terminal Step**	-1	Continues (`done` = `False`)

4 Agents and Learning Algorithms

In this project, three different agents were implemented to study and compare how various reinforcement learning strategies behave in the custom Gridworld environment. These include the Q-Learning agent, the SARSA agent, and a Naive Greedy agent used as a non-learning baseline. All learning agents use tabular action-value methods, with each state–action pair stored in a Q-table initialized to zero.

4.1 Q-Learning (Off-Policy TD Control)

Q-Learning is an off-policy temporal-difference method, meaning that it learns the value of the greedy target policy while following a potentially different exploratory behaviour policy (in our case, the ϵ-greedy policy). The update rule used in our implementation is:

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

Key characteristics of Q-Learning:

- It selects actions using an ϵ-greedy policy but updates using the greedy action for the next state.

- Being off-policy, it tends to converge faster to an optimal policy, especially in deterministic environments.
- However, due to its reliance on the maximization operator, it may exhibit instability when the exploration rate is high, especially near terminal pitfalls (holes).

4.2 SARSA (On-Policy TD Control)

SARSA is an on-policy learning algorithm that updates its action-value estimates using the action actually taken by the current behaviour policy. The update rule implemented in this project is:

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

Characteristics of SARSA:

- The method is "on-policy" because it evaluates and improves the same policy used to generate behaviour.
- SARSA naturally accounts for exploration in its updates, making it more conservative and stable.
- In our Gridworld, SARSA tends to avoid trajectories that pass too close to holes, learning safer policies compared to Q-Learning.

4.3 Naive Greedy Agent (Baseline)

The Naive Greedy agent serves as a non-learning baseline. It does not maintain a Q-table and performs no updates. Instead:

- It always selects the action with the highest immediate reward or progress toward the goal.
- It performs no exploration and has no ϵ parameter.
- Since it cannot learn from experience, its behaviour remains static across all episodes.

This baseline is useful for illustrating the necessity of learning and exploration in achieving convergence in stochastic or partially unknown environments.

5 Hyperparameters

The learning behaviour of Q-Learning and SARSA agents depends on several hyperparameters, all of which are fixed or dynamically adjusted as shown in Table 2.

For larger grid sizes (such as 7×7 or 10×10), the environment becomes more complex and requires longer exploration. The implementation automatically reduces the decay rate of ϵ for these larger grids to prevent premature convergence to suboptimal policies.

Hyperparameter	Symbol	Value	Purpose
Learning Rate	α	0.1	Controls the magnitude of updates to the Q-values.
Discount Factor	γ	0.95	Determines the importance of future rewards.
Exploration Rate	ϵ	1.0 (initial)	High initial exploration ensures sufficient coverage of the state–action space.
Exploration Decay	ϵ_{decay}	0.001	Governs how quickly exploration reduces over episodes.
Minimum Exploration	ϵ_{min}	0.01	Prevents the agent from becoming fully greedy too early.
Maximum Steps per Episode	—	$\text{gridSize}^3 + 200$	Ensures episodes terminate even if the agent loops indefinitely.

Table 2: Hyperparameters used in training the agents.

6 Training Procedure

The training engine operates in an interleaved manner to ensure a fair comparison. For every training iteration, the system sequentially executes one complete episode for the Q-Learning agent, one for the SARSA agent, and one for the Naive agent. This ensures that all agents are evaluated under identical exploration decay conditions and training duration.

6.1 Stopping Conditions

The training loop is designed with flexible termination criteria to accommodate varying grid complexities. The logic is handled within the `startTraining()` function:

- **Dynamic Episode Limit**: To prevent infinite loops while allowing sufficient learning time, the maximum number of episodes scales with the grid size.
  - For grids smaller than 10×10, training stops at **10,000 episodes**.
  - For larger grids, the limit extends to **20,000 episodes** to account for the larger state space.
- **Batch Mode**: Users can enable "Batch Mode" to execute a specific number of episodes (e.g., 200) per click, allowing for step-by-step analysis.
- **Manual Control**: An interactive Pause/Resume toggle allows the user to halt training instantly to inspect policy arrows and resume without resetting learned Q-values.

6.2 Performance Tracking

The system monitors four specific metrics in real-time to evaluate agent performance. Note that the system tracks both effectiveness (rewards) and efficiency (steps):

- **Episode Rewards**: The total accumulated reward per episode is stored in rolling history buffers (`q_rewards`, `sarsa_rewards`) to plot the learning curve.

- **Steps per Episode**: The number of steps taken to reach a terminal state is tracked. This metric is crucial for distinguishing between Q-Learning (which optimizes for the shortest path) and SARSA (which often takes longer, safer paths).
- **Dynamic Convergence Detection**: Instead of a hardcoded threshold, the code calculates convergence dynamically based on the grid size. An agent is considered "converged" if:
  1. The recent Success Rate exceeds 90%.
  2. The Average Reward exceeds 80% of the theoretical perfect run score (defined as $R_{\text{goal}} - \text{Minimum Steps}$).

7 Implementation Architecture

The system is implemented in **JavaScript** for client-side execution, organized into four modular components for clarity and maintenance.

Table 3: Modular Implementation Summary

Module	Core Responsibilities
Environment	Encapsulates Gridworld logic: • State management (1D index from $N \times N$ grid). • Deterministic transitions with boundary checks via `step(state, action)`. • Returns rewards ($+$**1000** Goal, $-$**100** Hole, $-$**1** Step).
RL Core	Manages agent memory and decision-making: • Stores Q-values in $\mathbf{3 \times 2D}$ arrays (Q-Learning, SARSA, Naive). • Implements ϵ-greedy action selection (with random tie-breaking). • Hyperparameters $(\alpha, \gamma, \epsilon)$ are reactive via UI sliders.
Training Engine	Executes the non-blocking learning loop: • Uses `requestAnimationFrame` for smooth, non-blocking UI updates. • Processes batches of episodes. • Manages and updates history buffers for analytical charts.
Visualization & Analytics	Presents real-time results and learned policies: • **Policy Overlay**: Renders the greedy action ($\arg\max Q$) on the grid. • **Charts** (Chart.js): Plots **Average Reward** (Effectiveness) and **Steps per Episode** (Efficiency).

8 Experiments and Results

8.1 Experimental Setup

To evaluate the scalability of the algorithms, we conducted experiments on a larger, more complex environment using the following configuration:

- **Grid Size**: We selected 13×13 (169 States) for our analysis.
- **Hyperparameters**: Adjusted for large-scale traversal:
  - Learning Rate (α): 0.15
  - Discount Factor (γ): 0.995 (High value to propagate rewards across long distances)
  - Exploration Decay: 0.0001 (Slow decay to ensure thorough exploration)

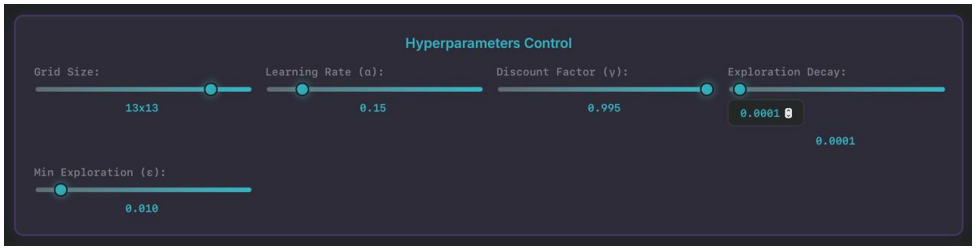

Figure 1: Grid initiation

  - **Holes**: We manually selected holes as shown in fig

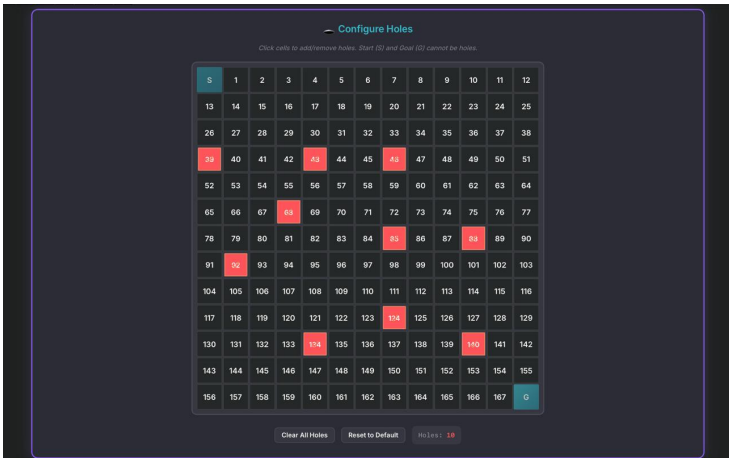

Figure 2: Configuring holes

  - **Episodes**: 20,000 (Increased to allow convergence on large grid).

8.2 Reward Analysis (Effectiveness)

The Average Reward chart (Figure 3) highlights a significant performance gap during the first half of training:

- **SARSA (Orange)**: Demonstrates superior stability. It achieves positive rewards rapidly (around episode 3,000). By penalizing dangerous exploration steps immediately, SARSA quickly learns a "safe" path, avoiding the negative 100 penalties from holes.
- **Q-Learning (Blue)**: Lags significantly. Because Q-Learning attempts to optimize for the mathematical shortest path (which often skims the edges of hole clusters), the high exploration rate causes frequent deaths. It only catches up to SARSA's performance after episode 9,000 when the exploration rate has sufficiently decayed.
- **Naive Agent**: Flatlines at the bottom, proving that random walking is impossible on a 13×13 grid.

Figure 3: Average Reward (13×13 Grid). SARSA stabilizes much earlier than Q-Learning.

8.3 Path Efficiency (Steps per Episode)

The Steps chart (Figure 4) reveals a fascinating "Safety Spike" phenomenon specific to SARSA on large maps:

- **The SARSA Spike**: Between episodes 2,000 and 6,000, SARSA's step count spikes dramatically (reaching ≈ 65 steps). This indicates that SARSA learned to take a massive detour—walking far around the hazards to ensure safety. As training progresses, it slowly tightens this path.
- **Q-Learning**: Maintains a consistently lower step count (≈ 30 steps) throughout. It refuses to take the long detour, preferring to attempt the risky shortcut repeatedly until it masters it.
- **Convergence**: By episode 20,000, both algorithms converge to a similar efficiency (≈ 28 steps), but Q-Learning arrives there by risking death, while SARSA arrives there by refining a safe route.

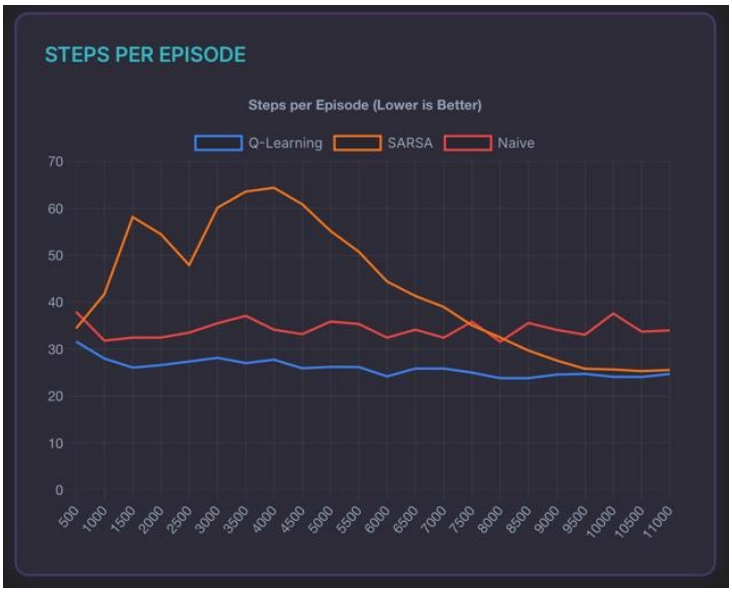

Figure 4: Steps per Episode. Note the large spike in SARSA's steps as it learns a safe detour, while Q-Learning maintains a risky but efficient path.

8.4 Convergence Analysis

Using the dynamic convergence threshold tailored for the 13×13 grid:

- **SARSA** converged significantly earlier (Episode 6,000 - 6,900). Its preference for safety allowed it to stop failing episodes quickly.
- **Q-Learning** converged much later (Episode 9,300 - 10,000). It required a much lower exploration rate (ϵ) before its risky optimal path became stable enough to meet the success criteria.

Figure 5: Convergence Analysis. SARSA solves the environment thousands of episodes before Q-Learning.

8.5 Policy Visualization

The final grid states (Figures 6 ,7 and 8) visualize the strategy difference:

- **Q-Learning**: Arrows generally point in the most direct line towards the goal, skimming the edges of the red hole clusters.
- **SARSA**: Arrows often show a wider berth around hazards, prioritizing cells that have zero risk of slipping into a hole.

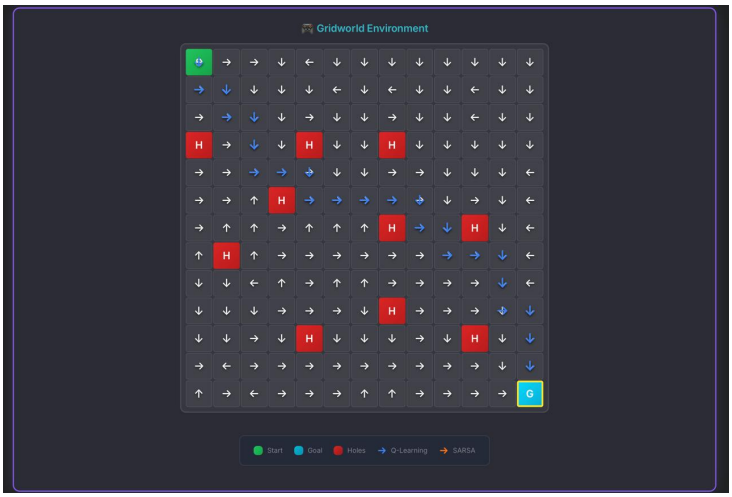

Figure 6: Q-Learning Policy

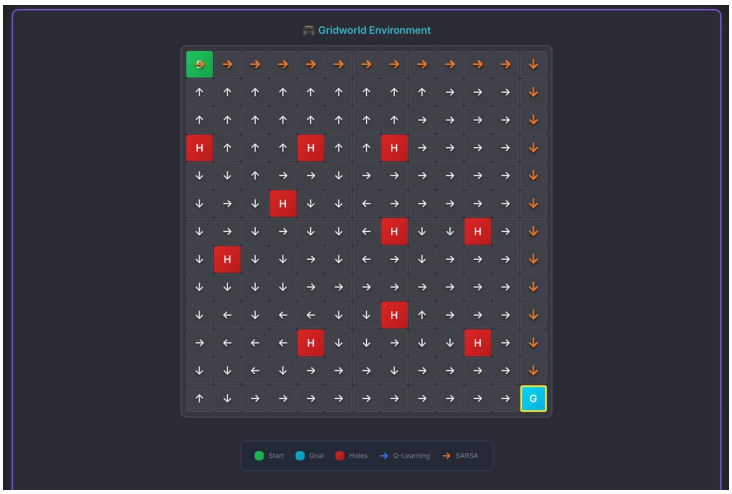

Figure 7: SARSA Policy

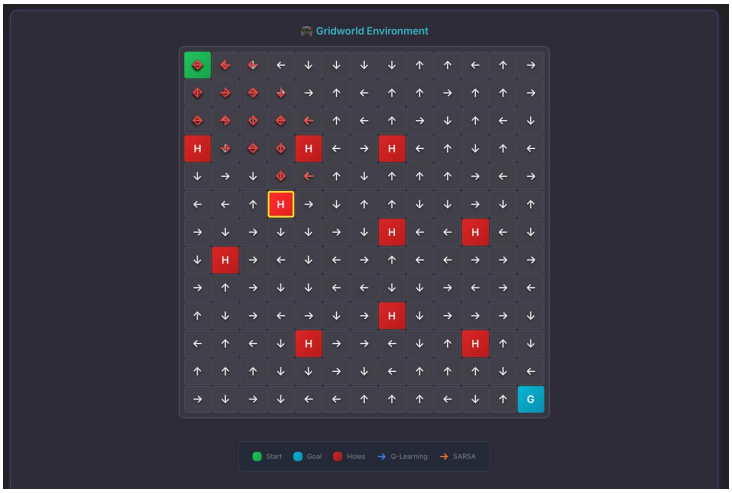

Figure 8: Naive

9 Discussion

The experimental results from the 13×13 grid provide a clear demonstration of the "Cliff Walking" phenomenon, highlighting the fundamental trade-off between on-policy and off-policy learning.

9.1 The Safety vs. Efficiency Trade-off

Q-Learning (The Optimistic Shortcut) As an off-policy algorithm, Q-Learning approximates the optimal policy q^*. It aggressively attempts to traverse the shortest path through the hole clusters.

- **Behavior**: The "Steps" chart shows Q-Learning maintaining a low step count (≈ 30) throughout training. It refuses to take a detour.
- **Consequence**: Because this optimal path leaves no margin for error, the high exploration rate during early training causes frequent deaths. This explains why Q-Learning's *Average Reward* lags behind SARSA for the first 9,000 episodes. It sacrifices safety for theoretical optimality.

SARSA (The Realistic Detour) As an on-policy algorithm, SARSA optimizes the value of the policy it is *actually* executing, including the exploration noise (ϵ).

- **Behavior**: The most distinct finding in the 13×13 experiment is the "Safety Spike" in the step count (Episodes 2,000–6,000). SARSA learned to take a massive detour (up to 65 steps) to skirt around the hazards entirely.
- **Consequence**: By taking the long way around, SARSA stopped falling into holes much earlier than Q-Learning. This allowed it to achieve a stable, positive reward signal thousands of episodes before Q-Learning, proving it is a more robust learner in hazardous environments.

9.2 Scalability and Convergence

The move to a larger state space (169 states vs. 16) magnified the behavioral differences. On the small grid, the "safe" path was only 1-2 steps longer. On the large grid, the safe path was nearly double the length of the optimal path. This highlights that while Q-Learning is superior for finding the mathematical limit of a deterministic system, SARSA is preferable for systems where failure is costly ,as it naturally incorporates a safety buffer.

10 Conclusion

This project implemented a modular Reinforcement Learning system to visualize the dichotomy between Optimality and Safety. By comparing Q-Learning and SARSA on complex grid topologies, we derived two key conclusions:

1. **Efficiency**: Q-Learning consistently identifies the shortest path (Step Count ≈ 28), regardless of the risk involved.
2. **Stability**: SARSA prioritizes survival. It achieved convergence criteria significantly faster (by Episode 6,000) compared to Q-Learning (Episode 9,500) because it quickly abandoned risky routes that led to negative terminal states.

The inclusion of a Naive baseline confirmed that in large state spaces (13×13), random traversal is statistically impossible, validating the necessity of TD-learning updates. The system successfully provides real-time, interactive proof of these fundamental RL concepts.

11 Future Work

To extend the capabilities of this framework, the following enhancements are proposed:

- **Stochastic Transitions (Wind)**: Introducing a probability of slipping (e.g., moving Right when Up is chosen) would further punish Q-Learning's risky behavior and highlight SARSA's robustness.
- **Deep Q-Networks (DQN)**: Replacing the tabular Q-table with a neural network would allow the agent to generalize across states, enabling the solution of much larger grids (e.g., 50×50) where tabular methods consume too much memory.
- **Dynamic Obstacles**: Introducing moving enemies would require the agents to learn temporal patterns rather than just static spatial paths.

12 Link to Our Webpage

You can explore our interactive Gridworld environment at the following link:
gridworld-rl-agent

13 Team Contributions

All team members contributed equally to the final report writing, debugging, and system testing. Specific technical implementations were divided as follows:

- **Revathi Katta (23110159):** Developed the core environment logic, state transitions, and dynamic hole configuration system.
- **Pappala Sai Keerthana (23110229):** Implemented the Q-Learning, SARSA, and Naive algorithms, along with the training loop logic.
- **Ravi Bhavana (23110274):** Built the visualization layer, including grid rendering, policy animations, and performance charts.
- **Thoutam Dhruthika (23110340):** Conducted experimental runs, recorded metrics, and synthesized the results for analysis.