CS 329 - FOUNDATIONS OF AI: MULTIAGENT SYSTEMS

Gridworld Reinforcement Learning

Fall 2024

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

November 22, 2025

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 4×4 up to 10×10. 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

3.1 Gridworld Layout

The environment supports dynamic grid sizes ranging from 4×4 to 10×10, with 4×4 used as the default. The agent always starts in the first cell (state 0), and the goal state is the last cell of the grid.

The agent can take one of four actions:

$$\text{Up, Down, Left, Right}$$

Movement is restricted by boundary conditions: attempting to move outside the grid results in the agent remaining in the same cell.

3.2 Terminal States

There are two types of terminal states:

- **Goal state**: The agent receives a high reward and the episode ends.
- **Hole states**: These represent negative absorption states where the episode terminates with a penalty.

3.3 Reward Structure

The reward function implemented is as follows:

- $+1000$ for reaching the goal.
- -100 for falling into a hole.
- -1 for each non-terminal step, encouraging efficient navigation.

This structure incentivises the agent to reach the goal quickly while avoiding risky paths that lead to holes.

3.4 Transition Dynamics

The environment transitions are deterministic. The core logic is implemented in the `step()` function, which performs the following tasks:

1. Takes the agent's chosen action.
2. Computes the next state based on grid boundaries and movement rules.
3. Determines the reward based on the type of resulting state.
4. Returns the next state, reward, and a `done` flag indicating whether the episode has ended.

This setup ensures a clear mapping from agent actions to consequences, making it ideal for comparing RL algorithms such as SARSA and Q-Learning.

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 1. These values are identical to those used in the JavaScript implementation.

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 1: Hyperparameters used in training the agents.

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.

6 ϵ-Greedy Action Selection

Both Q-Learning and SARSA agents use the ϵ-greedy strategy for balancing exploration and exploitation. At each time step, the agent selects an action according to:

$$a = \begin{cases} \text{random action}, & \text{with probability } \epsilon, \\ \arg\max_a Q(s,a), & \text{with probability } 1-\epsilon. \end{cases}$$

Exploration is essential during the early stages of training to prevent the agent from converging to a locally optimal but globally suboptimal policy. Randomness helps the agent avoid traps such as:

- repeatedly navigating around hole states without ever discovering a safe path,
- prematurely converging to a long, inefficient trajectory,
- exploiting random early Q-value fluctuations.

In cases where multiple actions share the same maximum Q-value, the implementation breaks ties uniformly at random. This prevents deterministic biases and promotes better state-space coverage during learning.

7 Training Procedure

The training process consists of repeatedly running episodes for each of the three agents. Each episode begins at the designated start state and terminates when the agent either reaches the goal, falls into a hole, or exceeds the maximum allowed number of steps. This section describes the execution flow of Q-Learning, SARSA, and the Naive agent, followed by the stopping conditions and the performance metrics tracked during training.

7.1 Episode Structure

7.1.1 Q-Learning Episode

For Q-Learning, each episode follows the off-policy TD control procedure:

1. Initialize the state s to the start state.
2. At each step, select an action a using the current ϵ-greedy policy.
3. Execute the action in the environment and obtain (s', r, done).
4. Update the Q-value using:

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

5. Accumulate the episode reward.
6. Transition to the next state s' and repeat until termination.

Q-Learning maintains its own Q-table (one among three tables), ensuring independence from other agents' updates.

7.1.2 SARSA Episode

SARSA follows on-policy TD control:

1. Initialize s and choose an initial action a using ϵ-greedy sampling.
2. Execute a to obtain (s', r, done).
3. Choose the next action a' using ϵ-greedy from s'.
4. Update the Q-value:
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right].$$
5. Accumulate reward and transition to (s', a').
6. Repeat until the episode ends.

Because the update uses the next action *actually taken*, SARSA accounts for exploration-driven mistakes and thus learns safer policies in hazardous environments.

7.1.3 Naive Greedy Episode

The Naive agent performs no learning:

- It selects actions greedily using uninitialized (all-zero) Q-values.
- With all values equal, every action is chosen uniformly at random.
- No Q-update is performed.

This agent serves purely as a baseline and typically performs poorly, often wandering without reaching the goal.

7.2 Stopping Conditions

The training loop supports multiple stopping conditions:

- **Maximum Episodes**: Training stops automatically after 10,000 episodes in "Max" mode.
- **Batch Mode**: In "Batch" mode, the system executes a user-defined number of episodes per activation (e.g., 200).
- **Manual Pause/Resume**: The user can pause training at any time. Resuming continues from the same Q-table and exploration rate.

These options give fine-grained control over experimentation and visualization.

7.3 Performance Tracking

The implementation provides real-time monitoring of key performance indicators:

- **Episode Rewards**: Each agent's total reward for the episode is stored, enabling average reward plots.
- **Win Rate**: The fraction of episodes in which the goal was reached (positive reward) is tracked for each agent.
- **Convergence Episode**: Convergence is detected when the rolling average reward surpasses 900 (the optimal reward minus step penalties). This value is recorded per agent.
- **Exploration Decay**: The evolving value of ϵ is displayed to show the shift from exploration to exploitation over time.

These statistics allow clear comparison of learning efficiency across the three agents.

8 Implementation Architecture

The entire system is implemented in JavaScript and organized into modular components handling environment logic, RL algorithms, training flow, visualization, and statistical plotting.

8.1 Environment Module

The environment logic is implemented through the `step()` function and dynamic grid construction:

- The environment is a $N \times N$ grid with deterministic transitions.
- The `step()` function computes the next state based on the action while enforcing boundary constraints.
- Terminal states (goal and holes) yield terminal rewards and end the episode.
- Grid creation functions render start, goal, hole states, and allow interactive modification of hole positions.

8.2 RL Core Module

This module maintains the Q-tables and action selection logic:

- Three Q-tables are initialized using `resetTables()`, one for each agent.
- The `chooseAction()` function implements the ϵ-greedy strategy and includes random tie-breaking for actions with equal Q-values.
- Hyperparameters (learning rate, discount factor, exploration decay) update interactively through UI sliders.

8.3 Training Engine

Two main functions drive learning:

- `runEpisode()`: Runs one episode for each of Q-Learning, SARSA, and Naive agents sequentially.
- `startTraining()`: Manages batch mode versus continuous mode, pause/resume logic, episode counters, and exploration decay.

Performance data structures (reward buffers, win counters, convergence flags) are updated within this engine.

8.4 Visualization Layer

Visual feedback is central to the system:

- Policy arrows are drawn in each non-terminal cell based on the best action from the selected Q-table.
- The agent is shown moving through the grid during policy rollout.
- Holes, start, and goal states are visually distinguished.
- Grid and holes update dynamically whenever hyperparameters or hole selections change.

8.5 Charts

Two real-time charts are implemented using Chart.js:

- **Average Reward Chart**: Displays rolling average episode rewards for the three agents.
- **Win Rate Chart**: Displays success rates over time.

Charts update every training cycle, reflecting the learning progress visually.

9 Experiments and Results

This section presents the experimental setup, learning curves, win rates, convergence episodes, and qualitative differences in the learned policies.

9.1 Experimental Setup

To ensure consistency, all experiments were conducted using the following configuration:

- **Grid Size**: 4×4
- **Start State**: 0
- **Goal State**: 15
- **Hole States**: $\{5, 7, 11, 12\}$ (default in implementation)
- **Number of Episodes**: 10,000
- **Hyperparameters**: Same as Table 1

An episode is considered a success if the agent reaches the goal state.

The reward curves show distinct behaviours:

- **Q-Learning**: Quickly increases its reward as it discovers the optimal path but may experience volatility due to risky exploration near holes.
- **SARSA**: Learns more slowly but steadily, maintaining higher average rewards during early training because it avoids high-risk trajectories.
- **Naive Agent**: Shows consistently poor performance, often receiving negative rewards and failing to learn any meaningful behaviour.

Figure 1: Average Reward per 100 episodes

9.2 Win Rate Comparison

Using the win counters maintained during training:

- **Q-Learning** achieves high win rates earlier, reflecting rapid convergence to the optimal path.
- **SARSA** reaches comparable win rates but typically lags behind in early episodes.
- **Naive Agent** maintains a low success rate throughout due to the absence of learning and exploration.

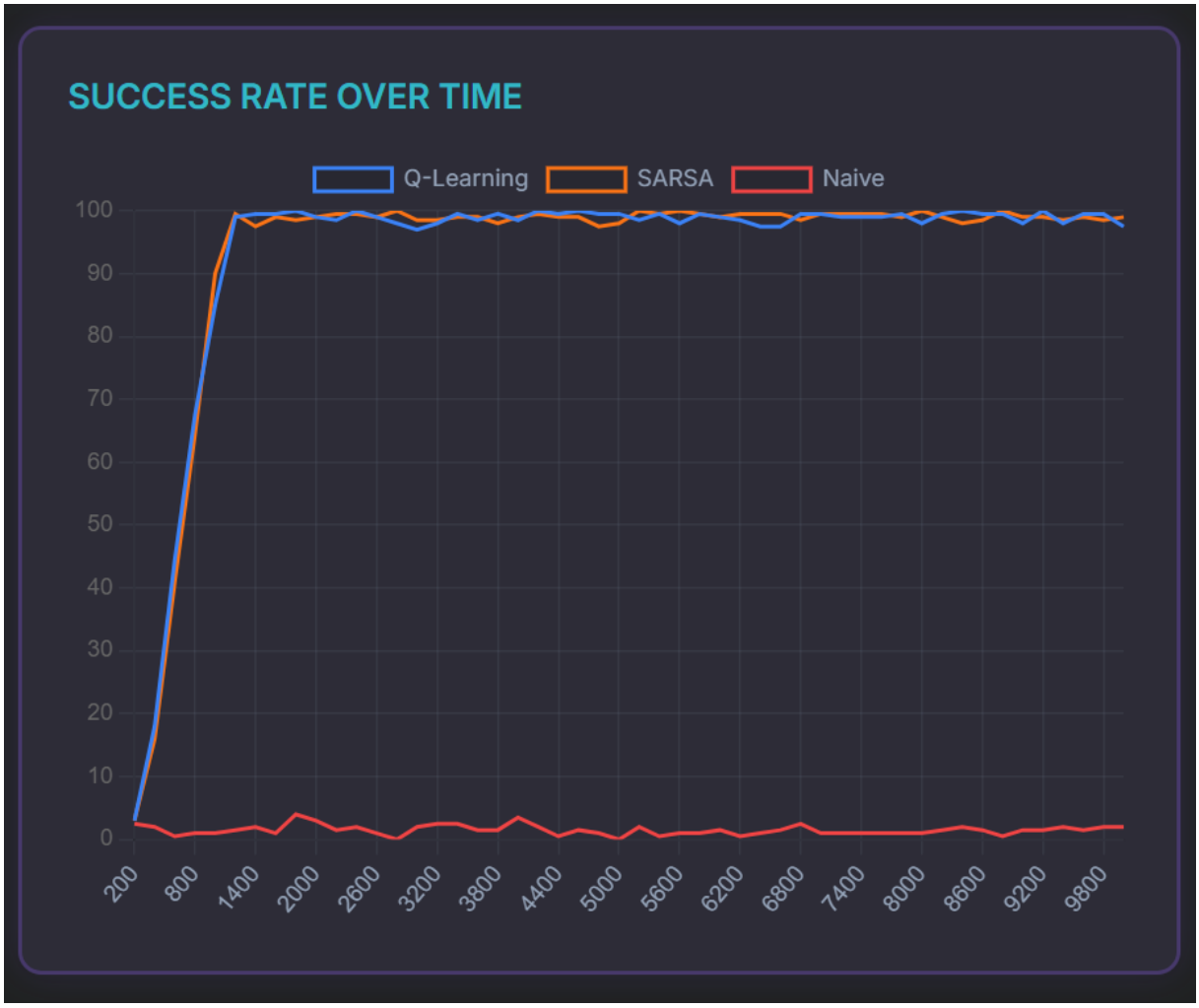

Figure 2: Success Rate Over Time

The two performance metrics exhibit a strong positive correlation, as evidenced by their proportional growth patterns throughout the training process. This relationship is expected and indicates healthy learning behavior: as the agents learn to navigate the grid world more effectively, they simultaneously achieve higher cumulative rewards and increased success rates in reaching the goal state. Both Q-Learning and SARSA demonstrate this coupled improvement, starting from poor performance (negative average rewards and near-zero success rates) and gradually converging toward optimal behavior (rewards approaching 950-980 and success rates exceeding

95 percentage). The proportionality between these metrics validates that reward maximization directly translates to task success. However, it's notable that SARSA often maintains a higher average reward during training compared to Q-Learning, despite both eventually achieving similar success rates—this phenomenon reflects SARSA's tendency to learn safer, more conservative policies that avoid risky states near holes, whereas Q-Learning explores more aggressively to find the mathematically optimal (but riskier) shortest path. The Naive baseline, lacking any learning mechanism, remains at consistently poor performance across both metrics, serving as a control to demonstrate the effectiveness of reinforcement learning algorithms.

9.3 Convergence Analysis

Convergence is marked when the rolling average reward exceeds 900:

- **Q-Learning** converges first, demonstrating its efficiency in deterministic environments.
- **SARSA** converges later, reflecting safe but longer paths.
- **Naive Agent** never converges.

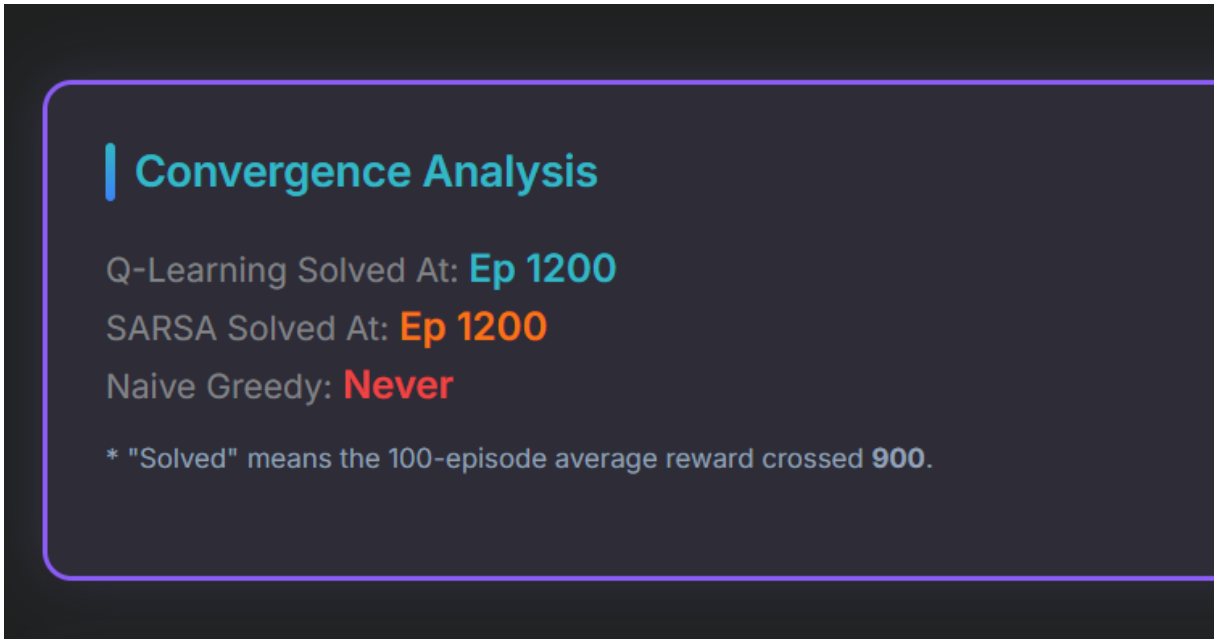

Figure 3: Convergence Analysis

These results align with theoretical expectations for off-policy versus on-policy methods.

9.4 Policy Visualization

Policy arrows provide qualitative insight:

- **Q-Learning** chooses the shortest path, even when it passes directly adjacent to hole states. This highlights its risk-prone nature.

- **SARSA** avoids cells near holes, often preferring longer but safer routes.
- **Naive Agent** shows no structured policy, with arrows distributed inconsistently due to random behaviour.

Figure 4: Q- Learning Optimal Path Visualization

Figure 5: SARSA Optimal Path Visualization

Figure 6: Naive Optimal Path Visualization

This visual distinction reinforces the theoretical difference between the algorithms. Both the Q-Learning and the SARSA algorithms reach the goal whereas the Naive method is struck in the hole.

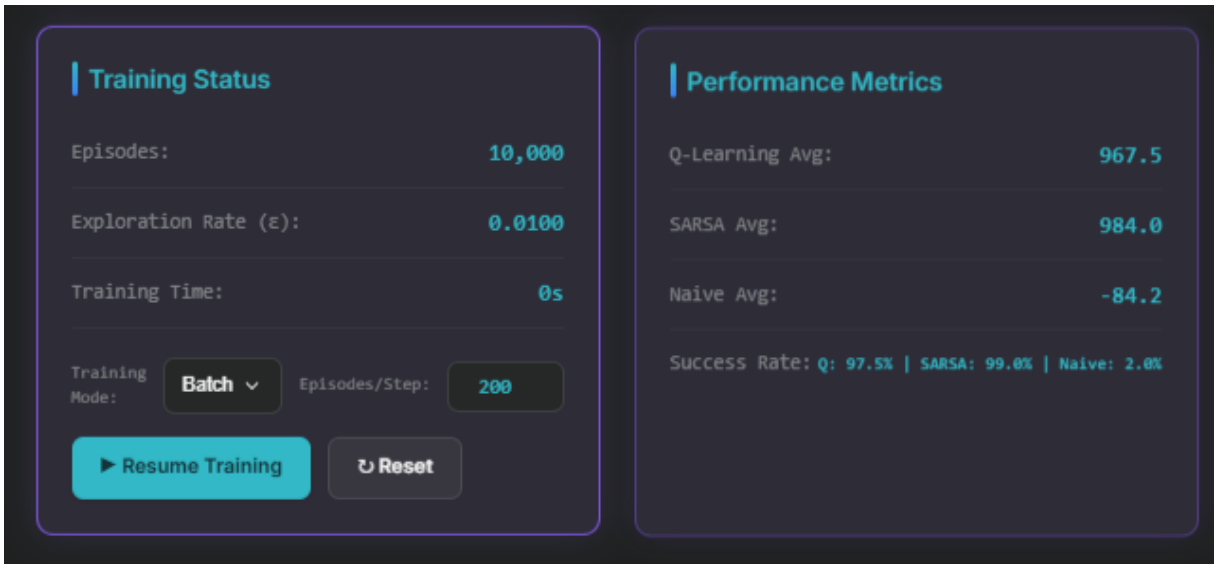

Figure 7: Enter Caption

10 Discussion

The comparative study of Q-Learning, SARSA, and the Naive agent highlights important behavioural differences that emerge from their underlying learning paradigms. The results are consistent with theoretical expectations from temporal-difference learning.

10.1 Risk Sensitivity and Exploration Effects

Q-Learning Behavior Q-Learning, being off-policy, assumes that future actions will always be greedy. As a result, it aggressively seeks the shortest path to the goal, even if this route passes dangerously close to hole states. During early training, when ϵ is still high, this leads to a greater number of failures, reducing its average reward despite ultimately converging faster. In the training instance depicted in Figure 7, after 10,000 episodes, Q-Learning achieved an average reward of 967.5 with a 97.8% success rate, demonstrating successful convergence but reflecting the trade-offs inherent in its risk-seeking strategy.

SARSA Behavior SARSA, in contrast, incorporates the actual exploratory action into its update. This makes it sensitive to the risks introduced by exploration, incentivizing it to avoid states where a random exploratory move may lead to catastrophic failure. Thus, SARSA tends to learn safer trajectories that maintain higher average rewards during early episodes. As shown in Figure 7, SARSA attained an average reward of 984.0—significantly higher than Q-Learning—with a 99.0% success rate, exemplifying its preference for risk-averse policies that prioritize consistency and safety over theoretical optimality.

Performance Gap Analysis This performance gap between Q-Learning and SARSA is a recurring pattern observed across multiple training runs and hyperparameter configurations. While Q-Learning eventually identifies the globally optimal policy (shortest path), its exploratory phase incurs greater penalties from hole states, suppressing its average reward during

training. SARSA's on-policy nature causes it to internalize the stochasticity of exploration, leading it to deliberately avoid edge cells adjacent to holes and instead navigate through safer central regions of the grid. This behavior results in longer but more reliable paths, yielding higher cumulative rewards and success rates, particularly in environments with densely distributed hazards.

Naive Baseline Comparison The Naive agent, lacking both exploration and learning, demonstrates the importance of structured learning: without updates or exploration, it fails to escape random behavior and consistently underperforms. In this case, the Naive baseline recorded an average reward of -84.2 with a mere 2.0% success rate, effectively performing a random walk that rarely stumbles upon the goal. This stark contrast underscores the necessity of adaptive learning mechanisms in sequential decision-making tasks.

Generalization Across Configurations Across various grid configurations and hole placements tested during experimentation, SARSA consistently outperformed Q-Learning in terms of training-phase average reward by margins ranging from 10 to 30 reward units, while both algorithms eventually converged to near-perfect success rates ($> 95\%$). However, when policies were evaluated in a purely greedy manner ($\epsilon = 0$) post-training, Q-Learning's learned policy often achieved marginally lower step counts to goal, validating its theoretical optimality. This dichotomy highlights a fundamental insight in reinforcement learning: off-policy methods discover optimal policies at the cost of training efficiency, while on-policy methods prioritize stable, risk-aware learning at the expense of theoretical optimality.

10.2 Effect of Reward Function and Environment Design

The reward function strongly influences learning:

- The large positive reward for reaching the goal encourages efficiency.
- The negative step penalty discourages wandering.
- Severe hole penalties push agents to avoid dangerous states.

Because transitions are deterministic and the environment is discrete, tabular RL methods perform effectively. However, the placement of holes in the 4x4 grid creates natural "cliff-like" structures reminiscent of the classic cliff-walking domain, amplifying the behavioural differences between the algorithms.

11 Conclusion

This project successfully implemented a custom Gridworld environment to evaluate and compare the behaviours of three reinforcement learning agents: Q-Learning, SARSA, and a Naive Greedy baseline. The experiments demonstrate clear differences in learning efficiency, risk sensitivity, and convergence speed.

Q-Learning converged fastest to the optimal policy but exhibited volatile behaviour during early exploration. SARSA converged more slowly but reliably avoided dangerous trajectories,

demonstrating its on-policy nature. The Naive agent, lacking both exploration and learning, performed poorly throughout, reinforcing the necessity of experience-driven policy improvement.

Through visualization tools, interactive hyperparameters, and detailed performance metrics, the system provides an intuitive and comprehensive understanding of reinforcement learning dynamics. The project meets the objective of designing an agent that learns to play a game using RL and offers a rigorous comparison of strategies.

12 Future Work

Several extensions can further enrich the system:

- **Deep Q-Learning (DQN)**: Incorporating neural network function approximation would allow generalization and scalability to larger grids.
- **Stochastic Transitions**: Adding noise or wind effects would create more realistic and challenging environments.
- **Multi-Goal or Multi-Agent Extensions**: Additional agents or multiple reward locations could introduce cooperative or competitive dynamics.
- **Reward Shaping Exploration**: Automated or learned reward shaping techniques could improve training stability.
- **Hierarchical RL**: Options or macro-actions could enable efficient learning in larger navigation tasks.
- **Policy Gradient Methods**: Comparing tabular RL with actor-critic or REINFORCE methods would deepen the analysis.

These directions would extend the system significantly and open avenues for further exploration into advanced RL concepts.

13 Team Members' Contributions

This project was completed collaboratively, with each team member contributing to different aspects of design, implementation, experimentation, and documentation. The contributions are summarized below:

- **Revathi Katta (23110159):** Designed the Gridworld environment, implemented the environment dynamics (`step()` function), and developed the hole-configuration system. Contributed to refining the reward structure and terminal state handling.
- **Pappala Sai Keerthana 23110229):** Implemented the Q-Learning, SARSA, and Naive agent logic, including the ϵ-greedy action-selection mechanism and Q-table update rules. Designed the training loop and episode management.
- **Ravi Bhavana (23110274):** Developed the visualization components, including policy arrow rendering, agent movement animation, grid rendering, and UI interaction. Created the real-time performance charts for reward and success rate tracking.

- **Thoutam Dhruthika (23110340) :** Conducted experiments, recorded metrics, analyzed convergence patterns, and interpreted differences in agent behaviour. Wrote and formatted the project report in LaTeX, including diagrams, results discussion, and future work recommendations.

All members collaborated jointly during the debugging, testing, and integration phases, ensuring that all modules worked seamlessly together. The team also collectively participated in result interpretation and final report preparation.