LLM Agent and Q-Learning for GridWorld: Implementation Guide

Thuong Dang

November 2025

Contents

1 Overview

This is the documentation for an experimental project: how an LLM agent thinks and plans in a simple gridworld game. Assume that we have a grid with empty cells, walls, the starting point and the target point. Our goal is to reach the target point from the starting point with the following rules:

- Valid moves include: left , right, up, down
- The penalty for an empty cell is -1
- The penalty for an obstacle is -5
- The reward for the target point is $+10$.

And our task is to maximize the reward on the grid. This project explores three questions:

1. How does an LLM agent reason about spatial navigation?
2. Can an LLM leverage algorithmic tools (RL, BFS) effectively?
3. Does an LLM fall for "trap" functions with misleading names?

2 The LLM Agent

2.1 Architecture

Our LLM agent is an LLM (GPT-4) equipped with a toolbox for navigation. The agent operates in a loop:

1. Observe current grid state and score
2. Reason about the situation
3. Select and execute one tool
4. Receive feedback (new position, reward, etc.)
5. Repeat until target reached or iteration limit

2.2 The Toolbox

The agent has access to 11 tools in `agent_tools.py`:

Direct movement:

- `move_up/down/left/right()`: Execute a move in that direction

Algorithmic helpers:

- `rl_move()`: Uses trained Q-learning policy (optimal if training succeeded)

- `bfs_move()`: Breadth-first search (finds shortest path, ignores rewards)

Exploration:

- `random_move()`: Random action
- `best_move()`: *Trap function* - claims to be optimal but returns random move! Tests if LLM blindly trusts tool names.

Look-ahead tools:

- `look_ahead_all()`: Preview all 4 directions without moving (once per game)
- `look_ahead_random()`: Preview one random direction (once per iteration)

Information:

- `get_current_state()`: Returns position, target, score, moves

2.3 System Prompt

The agent receives instructions via system prompt (`llm_agent.py`):

- Grid notation: `.` = empty, `X` = obstacle, `A` = agent, `T` = target
- Objective: Maximize score while reaching target
- Tool descriptions and usage limits
- Strategy hints: RL maximizes score, BFS finds shortest path, avoid obstacles

The agent must explain its reasoning before each tool choice, formatted as:

```
[reasoning] TOOL: tool_name
```

2.4 Agent Loop Implementation

Main loop (`llm_agent.py:run()`):

```
while not done and iteration < max_iterations:
    1. Reset iteration limits (look_ahead_random counter)
    2. Query LLM with current grid visualization
    3. Extract tool choice from LLM response
    4. Execute tool via AgentTools
    5. Display result and updated grid
    6. Add result to conversation history
    7. Check if target reached
```

Key points:

- Conversation history maintained across iterations (LLM can learn from mistakes)
- Tool results shown explicitly (encourages empirical reasoning)
- Resource limits enforced by `AgentTools` (strategic planning required)

3 GridWorld Environment

3.1 Grid Representation

The environment (`gridworld.py`) is an $n \times n$ grid (default: 10×10) represented as a numpy array:

- `0`: empty cell
- `-5`: obstacle cell
- Starting position: $(0, 0)$ (top-left)
- Target position: $(n-1, n-1)$ (bottom-right)

Obstacles are placed randomly at initialization, avoiding start and target positions.

3.2 State and Action Space

State space: $S = \{(x, y) : 0 \leq x, y < n\}$ (grid coordinates)

Action space: $A = \{0, 1, 2, 3\}$ where:

- 0 = up (decrease row)
- 1 = down (increase row)
- 2 = left (decrease column)
- 3 = right (increase column)

3.3 Dynamics and Rewards

The `step(action)` method implements deterministic transitions:

Reward function:

$$r(s, a) = \begin{cases} +10 & \text{if } s' = s_{\text{target}} \text{ (goal reached)} \\ -5 & \text{if } s' \text{ is obstacle or out of bounds} \\ -1 & \text{otherwise (empty cell)} \end{cases}$$

Boundary handling: Attempting to move out of bounds results in $r = -5$, agent stays at current position.

Termination: Episode ends when agent reaches target (`done` = `True`).

3.4 Key Methods

- `reset()`: Returns agent to start position, resets score and move counter
- `step(action)`: Executes action, returns $(s', r, \texttt{done})$
- `get_state()`: Returns current position as tuple (for Q-table indexing)
- `render()`: Prints grid with visual symbols (A=agent, T=target, X=obstacle, .=empty)
- `get_grid_string()`: Returns grid as formatted string (for LLM prompt)

4 Q-Learning Training

4.1 Overview

The Q-learning implementation consists of two files:

- `q_learning.py`: The `QLearningAgent` class implementing the Q-learning algorithm
- `train_rl.py`: Training script that creates environment, trains agent, and saves results

4.2 The QLearningAgent Class

4.2.1 Initialization

The agent is initialized with hyperparameters:

- `learning_rate` ($\alpha = 0.1$): Controls how much we update Q-values each step
- `discount_factor` ($\gamma = 0.9$): How much we value future rewards (must be < 1 for convergence!)
- `epsilon` ($\epsilon = 1.0$ initially): Exploration rate for ϵ-greedy policy
- `epsilon_decay` (0.995): Multiplicative decay applied after each episode
- `epsilon_min` (0.01): Minimum exploration rate

The Q-table is initialized as:

```
self.q_table = defaultdict(lambda: np.zeros(n_actions))
```

This means: for any state (x, y), the first time we visit it, automatically create an entry with Q-values $[0, 0, 0, 0]$ for the four actions. Only visited states are stored (memory efficient).

4.2.2 Exploration vs Exploitation: get_action() and decay_epsilon()

Action selection with ϵ-greedy:

$$a = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \text{argmax}_{a'} Q(s, a') & \text{with probability } 1 - \epsilon \end{cases}$$

Key point: ϵ-greedy is NOT part of the Q-learning algorithm itself! It's a separate exploration strategy that balances:

- **Exploration** (random actions): Discover new states and potentially better paths
- **Exploitation** (greedy actions): Use current knowledge to maximize reward

The `explore` flag:

- During training: `explore=True` uses ϵ-greedy to gather diverse experiences
- During testing/deployment: `explore=False` uses pure greedy (best learned policy)

Epsilon decay schedule: After each episode, we decrease exploration: $\epsilon \leftarrow \max(\epsilon_{\min}, 0.995 \cdot \epsilon)$

- Episode 1: $\epsilon = 1.0$ (pure exploration — discover all states)
- Episode 500: $\epsilon \approx 0.08$ (mostly exploiting learned policy)
- Episode 1000+: $\epsilon = 0.01$ (minimal exploration to avoid getting stuck)

Why? Early in training, explore to discover the environment. As the agent learns, shift toward exploiting the learned policy while maintaining 1% exploration.

4.2.3 Q-Value Update: update()

This is the **core of Q-learning**. After observing transition (s, a, r, s'):

```
def update(self, state, action, reward, next_state, done):
    current_q = self.q_table[state][action]

    if done:
        target_q = reward  # No future rewards at terminal state
    else:
        max_next_q = np.max(self.q_table[next_state])
        target_q = reward + self.gamma * max_next_q

    # Q-learning update rule
    self.q_table[state][action] = current_q + self.lr * (target_q - current_q)
```

What's happening:

1. **Compute target:** What should $Q(s, a)$ be?
   - If episode ended: target $= r$ (no future rewards)
   - Otherwise: target $= r + \gamma \max_{a'} Q(s', a')$ (immediate reward + discounted best future value)
2. **Compute temporal difference (TD) error:** $\delta = \text{target} - Q(s, a)$
   - Positive error: we underestimated, increase Q-value
   - Negative error: we overestimated, decrease Q-value
3. **Update Q-value:** $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \delta$
   - Small α: slow, stable learning
   - Large α: fast but noisy learning

Mathematical form:

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

This is exactly the equation from theory document Section 3.

Why this works: From theory Section 4, the Bellman operator is a γ-contraction. This update is a stochastic approximation of applying the Bellman operator. Over many updates, Q converges to Q^*.

4.2.4 Explicit calculation: First Update

Let's trace through a concrete example to see how the update works in practice. Consider we are at episode 1, step 1. The agent starts at $(0, 0)$. At the beginning, $\epsilon = 1$, so we explore randomly. Suppose we choose the action `right` and move into an empty cell with reward $r = -1$.

Before update:

- $Q((0, 0), \text{right}) = 0$ (initial value)
- $Q((0, 1), \cdot) = [0, 0, 0, 0]$ (all actions at next state are 0)

Observation:

- State: $s = (0, 0)$
- Action: $a = \text{right}$
- Reward: $r = -1$ (empty cell penalty)
- Next state: $s' = (0, 1)$
- Done: False

Update calculation with $\alpha = 0.1, \gamma = 0.9$

$$\text{current\_q} = Q((0,0), \text{right}) = 0$$
$$\max_{a'} Q((0,1), a') = \max(0,0,0,0) = 0$$
$$\text{target\_q} = r + \gamma \max_{a'} Q(s', a') = -1 + 0.9 \times 0 = -1$$
$$\text{TD error} = \text{target\_q} - \text{current\_q} = -1 - 0 = -1$$
$$Q((0,0), \text{right}) \leftarrow 0 + 0.1 \times (-1) = -0.1$$

After this update, $Q((0,0), \text{right}) = -0.1$, and the agent has learned that moving `right` from $(0,0)$ yields a small negative value, reflecting the cost of stepping into an empty cell. As training continues, ϵ gets smaller and smaller, and the agent relies more on its learned Q-values rather than exploring randomly. However, we maintain a small exploration probability ($\epsilon_{\min} = 0.01$) to avoid getting stuck in local optima.

4.2.5 Training Loop: train()

This orchestrates the entire learning process:

```
def train(self, env, n_episodes=5000, max_steps=200, verbose=True):
    episode_rewards = []

    for episode in range(n_episodes):
        state = env.reset()              # Start new episode
        total_reward = 0

        for step in range(max_steps):
            # 1. Choose action using -greedy
            action = self.get_action(state, explore=True)

            # 2. Execute action in environment
            next_state, reward, done = env.step(action)

            # 3. Update Q-table immediately after each transition
            self.update(state, action, reward, next_state, done)

            total_reward += reward
            state = next_state

            if done:
                break  # Episode finished (reached target)

        # 4. Decay epsilon after each episode
        self.decay_epsilon()
        episode_rewards.append(total_reward)
```

```
    return episode_rewards
```

Episode structure:

- One episode = one complete game from start to target (or timeout after 200 steps)
- Agent experiences many episodes to learn from diverse trajectories
- Each episode may have different length depending on path taken

Update timing: After EVERY transition (s, a, r, s'), we immediately call `update()`. We don't wait until episode ends. This allows the agent to learn from each step, resulting in faster convergence.

Progress tracking:

- `episode_rewards`: stores total reward for each episode
- Printed every 500 episodes: moving average (last 100 episodes) and current ϵ
- Used to plot training curve and verify convergence

4.2.6 Model Persistence

The agent can save and load trained Q-tables:

- `save()`: Serializes Q-table and hyperparameters to disk using pickle
- `load()`: Restores Q-table as a `defaultdict` for immediate use

This allows reusing trained agents without retraining, which is essential for the LLM agent's `rl_move()` tool.

4.3 Training Script: train_rl.py

4.3.1 Training Configuration

```
# Create fixed grid
env = GridWorld(size=10, num_obstacles=45)

# Create agent
agent = QLearningAgent(
    learning_rate=0.1,
    discount_factor=0.9,
    epsilon=1.0,           # Start with full exploration
    epsilon_decay=0.995,
    epsilon_min=0.01
)
```

```
# Train
rewards = agent.train(env, n_episodes=5000, max_steps=200)
```

Hyperparameter choices:

- $\alpha = 0.1$: Moderate learning rate (stable but not too slow)
- $\gamma = 0.9$: Values future rewards while ensuring $\gamma < 1$ for contraction (theory Section 4)
- $\epsilon_0 = 1.0$: Start with pure random exploration to discover all states
- 5000 episodes: Sufficient for convergence on 10×10 grid
- 45 obstacles on 10×10 grid: Creates challenging environment where strategic planning is necessary

4.3.2 Fixed Grid Training

The script creates ONE grid with randomly placed obstacles and trains on it for all 5000 episodes.

Why fixed grid?

- Standard RL setup: learn optimal policy for a specific environment
- Every state visited many times $\rightarrow$ Q-values converge
- Fair evaluation: LLM agent uses the same grid and trained Q-table

Note: This Q-table is specific to this grid layout. A different obstacle configuration requires retraining.

4.3.3 Outputs

- `models/q_table.pkl`: Trained Q-table (loaded by LLM agent's `rl_move()` tool)
- `models/grid_config.pkl`: Grid layout (ensures LLM uses same grid for testing)
- `models/training_progress.png`: Reward curve showing convergence over episodes

5 Experimental Results

The LLM agent for this experiment is GPT-4o. The environment is a 10×10 grid with 45 randomly placed obstacles. Note that the LLM plays the game *blindly*: it does not know the grid layout in advance. All it receives are the tool descriptions, the evolving game state, and feedback after each move, no global

map or internal memory of the grid beyond what it infers from interaction. Yet, even with such limited information, it has shown effective behavior. Here are the main observations:

1. **Always starts with** `look_ahead_all()`: Uses the one-time tool immediately to survey all directions.
2. **Delegates to algorithms**: Prefers `bfs_move()` or `rl_move()` over manual navigation.
3. **Adapts when hitting obstacles**: If BFS hits obstacles (reward = -5), switches to RL, and vice versa.
4. **Occasional manual control**: If the path is clear, for example, no obstacles to the right, the agent repeatedly uses `move_right`.
5. **Sometimes uses** `look_ahead_random()`: Gathers additional information when uncertain about the next move.
6. `best_move()` **never chosen**: Despite being the trap function, the LLM never selected it. This suggests a preference for tools with explicit descriptions (e.g., `bfs_move`, `rl_move`) over vague alternatives.
7. **Goal-reaching performance:** Across roughly 20 tests, the LLM agent almost always managed to reach the target. However, it rarely achieved the optimal reward, especially when the number of obstacles was high. This indicates competent planning under uncertainty, but not full optimality.

A key observation is that, even with very sparse metadata about the tools (e.g., no information of how accurate the RL model is), the LLM still uses algorithmic helpers strategically and it suggests good reasoning capabilities.

Running this system with larger models becomes expensive. Until a game ends, the agent needs the full conversation history (including tool descriptions reasoning, grid states for each iteration). If we scale up the number of tools or grid size, the input tokens grow significantly. Two questions are:

1. How does tool selection scale when the number of tools increases?
2. How can we help LLMs make better decisions when they lack global structure awareness?