# Q-Learning in a 5x5 Grid World: Implementation and Analysis

Emanda Hailu
GSR 5056/17

January 29, 2026

## 1 Overview

This report presents the implementation of a basic Q-learning algorithm for a simple $5 \times 5$ grid world environment. The agent can move in four directions (up, down, left, right), receives a reward of $+10$ for reaching the goal state, and a reward of $-1$ for every other step. The Q-table is initialized with zeros, and learning is performed with a learning rate $\alpha = 0.5$ and discount factor $\gamma = 0.9$. A fixed number of episodes is used for training, and the resulting Q-values are reported and discussed.

## 2 Introduction

Reinforcement learning (RL) is a framework in which an *agent* learns to make sequences of decisions by interacting with an environment and receiving feedback in the form of rewards. Among model-free RL methods, Q-learning is a widely used off-policy temporal-difference control algorithm that learns an action-value function $Q(s, a)$ without requiring a model of the environment's dynamics.

In this report, we consider a deterministic $5 \times 5$ grid world as the environment. The goal is to show how to implement Q-learning from scratch in Python, construct and update the Q-table, and interpret the learned Q-values after training.

## 3 Problem Formulation

### 3.1 Grid World Environment

We consider a grid world with 5 rows and 5 columns, for a total of 25 states. Each cell represents a state. The agent starts in the top-left corner of the grid, denoted as position $(0, 0)$ using zero-based indexing. The goal state is the bottom-right corner $(4, 4)$.

At each time step, the agent can choose one of four actions:

- Up: move from $(i, j)$ to $(i - 1, j)$,
- Down: move from $(i, j)$ to $(i + 1, j)$,
- Left: move from $(i, j)$ to $(i, j - 1)$,
- Right: move from $(i, j)$ to $(i, j + 1)$.

If the agent attempts to move outside the grid (e.g., moving up from the top row), the agent remains in the same state and still receives the step penalty.

### 3.2 Rewards and Episode Termination

The reward function is defined as:

- $+10$ when the agent transitions into the goal state $(4, 4)$,
- $-1$ for every other transition, including invalid moves that leave the agent in place.

An episode terminates as soon as the goal state is reached. During training, each episode starts with the agent in the initial state $(0, 0)$.

### 3.3 State and Action Representation

To implement the Q-table as a 2D NumPy array, we map 2D grid positions to a 1D state index:

$$s = i \times \text{grid\_size} + j, \tag{1}$$

$$(i, j) = \text{divmod}(s, \text{grid\_size}). \tag{2}$$

For a $5 \times 5$ grid, $s \in \{0, 1, \ldots, 24\}$. There are four discrete actions, which we index as:

$$a \in \{0, 1, 2, 3\},$$

corresponding to up, down, left, and right, respectively.

## 4 Q-Learning Algorithm

### 4.1 Q-Value Definition

The central object in Q-learning is the action-value function $Q(s, a)$, defined as the expected discounted return when starting from state $s$, taking action $a$, and thereafter following the optimal policy:

$$Q(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \,\middle|\, s_0 = s, a_0 = a, \pi = \pi^*\right].$$

### 4.2 Update Rule

Q-learning updates the Q-values iteratively using the temporal-difference rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right],$$

where:

- $\alpha \in (0, 1]$ is the learning rate,
- $\gamma \in [0, 1]$ is the discount factor,
- $r_{t+1}$ is the reward received after taking action $a_t$ in state $s_t$,
- $s_{t+1}$ is the resulting next state.

In our setup, we use $\alpha = 0.5$ and $\gamma = 0.9$.

### 4.3 Exploration Strategy

To balance exploration and exploitation, we use an $\varepsilon$-greedy policy:

- With probability $\varepsilon$, the agent chooses a random action.
- With probability $1 - \varepsilon$, the agent chooses the greedy action $\arg\max_a Q(s, a)$.

In the implementation, we use a fixed $\varepsilon = 0.1$.

### 4.4 Algorithm Pseudocode

A high-level pseudocode description for episodic Q-learning in this grid world is:

1. Initialize $Q(s, a) = 0$ for all states $s$ and actions $a$.
2. For each episode:
   (a) Set the current state $s$ to the start state.
   (b) Repeat until the goal state is reached:
      i. Select action $a$ using $\varepsilon$-greedy policy based on $Q(s, \cdot)$.
      ii. Execute action $a$, observe reward $r$ and next state $s'$.
      iii. Update:
$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right].$$
      iv. Set $s \leftarrow s'$.

## 5 Python Implementation

### 5.1 Environment and Helper Functions

Listing 1 defines the grid world parameters, state–position mappings, and the environment transition function `step`.

Listing 1: Grid world setup and environment dynamics

```
import numpy as np

# Grid world parameters
grid_size = 5
goal = (4, 4)  # bottom-right as goal

# Actions: 0-up, 1-down, 2-left, 3-right
actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
n_actions = len(actions)

# Initialize Q-table: 25 states x 4 actions
Q = np.zeros((grid_size * grid_size, n_actions))

# Hyperparameters
alpha = 0.5
gamma = 0.9
epsilon = 0.1
n_episodes = 1000

def state_to_pos(s):
    """Convert state index to grid coordinates (i, j)."""
    return divmod(s, grid_size)

def pos_to_state(i, j):
    """Convert grid coordinates (i, j) to state index."""
    return i * grid_size + j

def step(state, action):
    """
    Environment transition function.
```

```
    Given a state index and action, returns next_state, reward, done.
    """
    i, j = state_to_pos(state)
    di, dj = actions[action]
    ni, nj = i + di, j + dj

    # Check grid boundaries
    if 0 <= ni < grid_size and 0 <= nj < grid_size:
        next_state = pos_to_state(ni, nj)
        reward = 10 if (ni, nj) == goal else -1
    else:
        # Invalid move: stay in place
        next_state = state
        reward = -1

    done = (ni, nj) == goal
    return next_state, reward, done
```

### 5.2 Training Loop with Q-Learning

Listing 2 shows the main training loop that applies the Q-learning update for a fixed number of episodes.

Listing 2: Q-learning training loop for the 5x5 grid world

```
for ep in range(n_episodes):
    # Start each episode at the top-left corner (state 0)
    state = 0

    while True:
        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            action = np.random.randint(n_actions)  # explore
        else:
            action = np.argmax(Q[state])           # exploit

        # Take action and observe next state and reward
        next_state, reward, done = step(state, action)

        # Compute the best next Q-value
        best_next = np.max(Q[next_state])

        # Q-learning update
        Q[state, action] += alpha * (
            reward + gamma * best_next - Q[state, action]
        )

        # Move to the next state
        state = next_state

        # End episode if goal is reached
        if done:
            break

# Print the learned Q-table after training
print("Learned Q-values (25 states x 4 actions):")
print(Q)
```

### 5.3 Learned Q-Values

After training for 1000 episodes, the Q-table contains the estimated action-values for each state–action pair. One possible Q-table (values will vary due to randomness) is:

```
[[-1.39065580e+00 -4.34071407e-01 -1.39065583e+00 -4.34062000e-01]
 [-4.34062081e-01  6.28820000e-01 -1.39066007e+00  6.28817564e-01]
 [-1.85493750e+00  1.80979976e+00 -6.65094875e-01 -1.07161252e+00]
 [-1.42625000e+00  2.84406249e+00 -1.81268750e+00 -1.64118750e+00]
 [-1.85493750e+00  3.04998039e+00 -1.69062500e+00 -1.85493750e+00]
 [-2.61804008e+00 -2.33329687e+00 -1.00615684e+00  6.28820000e-01]
 [-4.34062000e-01  1.80980000e+00 -4.34062208e-01  1.80979980e+00]
 [-5.54027769e-01 -1.53937500e+00 -4.46808750e-01  3.12200000e+00]
 [ 6.13312483e-01  4.58000000e+00  3.61150000e-01 -9.75000000e-01]
 [-7.37711949e-01  6.19999913e+00 -1.19548437e+00  1.57687187e+00]
 [-2.12150625e+00 -1.94343750e+00  1.48154628e-03  1.80980000e+00]
 [ 6.28757284e-01  3.12190443e+00  6.28819987e-01  3.12200000e+00]
 [ 1.80968976e+00  4.57982583e+00  1.80979995e+00  4.58000000e+00]
 [ 3.12199997e+00  6.19994679e+00  3.12199908e+00  6.20000000e+00]
 [ 4.57998128e+00  8.00000000e+00  4.57999999e+00  6.20000000e+00]
 [-2.03638963e+00 -1.88618750e+00 -1.65187500e+00 -1.70812500e+00]
 [ 1.43681250e-01 -1.69687500e+00 -1.48250000e+00  4.57999610e+00]
 [ 9.04125000e-01 -1.10000000e+00  1.01702581e+00  6.20000000e+00]
 [ 4.40640625e+00  8.00000000e+00  1.92610179e+00  6.93750000e+00]
 [ 6.20000000e+00  1.00000000e+01  6.19999885e+00  7.99999952e+00]
 [-1.82681250e+00 -1.85493750e+00 -2.06928125e+00 -1.80937500e+00]
 [-1.78687500e+00 -1.80937500e+00 -1.69118750e+00 -1.37562500e+00]
 [-9.75000000e-01 -9.75000000e-01 -1.18875000e+00  7.58590317e+00]
 [ 1.99218750e+00  3.74890137e+00  4.27646980e+00  1.00000000e+01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
```

Each row corresponds to a state (from 0 to 24), and each column corresponds to an action (up, down, left, right). The goal state's Q-values remain zero because it is terminal and never updated.

## 6 Interpretation of the Learned Policy

A greedy policy can be extracted from the Q-table by selecting, for each state $s$, the action with the highest Q-value:

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

For states near the start, the optimal actions generally point toward the goal at $(4, 4)$, moving predominantly right and down. Near the goal, Q-values become large and positive for actions that quickly reach the terminal state, reflecting the future discounted reward of $+10$ minus the step costs.

Because the training process involves randomness due to $\varepsilon$-greedy exploration, running the algorithm multiple times can produce slightly different Q-tables, but the qualitative structure of the learned policy should remain consistent.

## 7 Conclusion

This report demonstrated a complete implementation of the Q-learning algorithm for a simple $5 \times 5$ grid world. Starting from an all-zero Q-table, the agent learns action values through

interaction with the environment, using a reward of $+10$ for reaching the goal and $-1$ otherwise. With a learning rate of $\alpha = 0.5$, discount factor $\gamma = 0.9$, and an $\varepsilon$-greedy exploration strategy, the agent quickly converges to a policy that moves efficiently from the start state to the goal state.

The code can be extended to visualize the policy, adjust hyperparameters, or compare Q-learning with other reinforcement learning algorithms such as SARSA or value iteration.