《人工智能原理》项目 3 报告
Q_learning 算法求解迷宫宝藏问题

摘要

本项目使用 Q-learning 和 2-step Q-learning 算法求解迷宫宝藏问题。首先对问题进行马尔可夫决策过程 (MDP) 的建模，定义状态空间、动作集合、状态转移概率和回报函数。随后实现标准 Q-learning 算法，并扩展为 2-step Q-learning，以利用多步回报信息加速学习。通过对比实验，分析了两种方法在不同参数设置下的表现差异，探讨了 N-step 方法在处理稀疏奖励和长程依赖问题上的优势与挑战。实验结果表明，2-step Q-learning 在引入步数惩罚的环境中表现出更快的收敛速度和更优的策略质量。
关键词：强化学习，Q-learning，马尔可夫决策过程

1 问题建模

1.1 状态空间

迷宫建模为 6×6 的二维网格。利用劳拉在迷宫中的网格坐标定义状态 S，所给问题中共有 36 个坐标，故状态空间：

$$S = \{(x, y) \mid 0 \leq x \leq 5, 0 \leq y \leq 5\}, |\mathcal{S}| = 36 \tag{1}$$

在代码实现中，将二维坐标 (r, c) 映射为一维索引 $0 \sim 35$。

$$\text{State\_Index} = r \times \text{Width} + c$$

未引入额外的虚拟终止状态，宝藏位置以及陷阱位置作为状态空间的一部分存在，同时被标记为终止态 (S_{terminal})，当智能体到达上述状态之一时，触发 `done=True`，回合立即结束、不再执行后续动作。

1.2 动作集合

根据题意，劳拉在每个状态下可以尝试向上下左右四个方向移动。故定义动作集合为：

$$A = \{\text{Up}, \text{Down}, \text{Right}, \text{Left}\} \tag{2}$$

在代码实现中，动作与整数映射关系：0: Up, 1: Down, 2: Right, 3: Left。

1.3 状态转移概率

环境是确定性的，且对智能体无模型的。对任意非终止状态 $s = (r, c) \in S$，执行动作 $a \in A$，状态将按照动作方向确定性地转移。状态转移规则为：

- 若 a 指向的方向无墙壁，则以概率 1 转移到相邻格子。
- 若 a 导致智能体移出迷宫边界即指向墙壁，则 $s' = s$（不动）。
- 特别的，若 s' 为宝藏位置或陷阱位置，触发 `done=True`，回合结束。

定义移动函数：

$$\text{move}(s, a) = (r', c')$$

其中：

$$(r', c') = \begin{cases} (r-1, c), & a = \text{Up},\ r > 0 \\ (r+1, c), & a = \text{Down},\ r < 5 \\ (r, c+1), & a = \text{Right},\ c < 5 \\ (r, c-1), & a = \text{Left},\ c > 0 \\ (r, c), & \text{otherwise(撞墙)} \end{cases}$$

状态转移结果定义为：$s' = \text{move}(s, a)$，并规定 $s' \in S_{\text{terminal}}$ 时回合终止。
则状态转移概率函数为：

$$P(s' \mid s, a) = P(S_{t+1} = s' \mid S_t = s, A_t = a) = \begin{cases} 1, & s' = f(s, a) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

其中 $f(s, a)$ 表示上述确定性状态转移规则。

1.4 回报函数

在 `Utils.reward_process` 中定义回报如下：

$$R(s, a, s') = \begin{cases} +5.0, & s' = \text{treasure(找到宝藏)} \\ -5.0, & s' \in \text{traps(掉入陷阱)} \\ -0.1, & s' = s\text{(撞墙)} \\ 0.0, & \text{otherwise(普通移动)} \end{cases} \tag{4}$$

2 Q-learning 算法实现

2.1 Q-table 定义

Q-table 被定义为一个二维 NumPy 数组，用于存储状态-动作价值。

```
self.q_table = np.zeros((n_states, n_actions))
```

其中维度 1 为状态数 36，维度 2 为动作数 4，初始化为全 0。每个状态-动作对应一个 $Q(s, a)$，表示在状态 s 下执行动作 a 的期望累积回报。

2.2 算法与参数设置

Q-Leaning 方法的行动价值更新公式为：

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left(R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right) \tag{5}$$

利用 ϵ-greedy 算法选择行为策略：

$$\begin{cases} \pi(a \mid s) = 1 - \epsilon + \frac{\epsilon}{m}, & \text{if } a = \arg\max_{a \in A} Q(s, a) \\ \pi(a \mid s) = \frac{\epsilon}{m}, & \text{otherwise} \end{cases} \tag{6}$$

其中 m 为动作数，α 为学习率，γ 为折扣因子，探索率 ϵ 用于平衡探索与利用。本次实验中设置：较小的 $\alpha = 0.1$；较大的 $\gamma = 0.9$ 以重视长期回报；ϵ 采用动态衰减策略，初始为 0.3，每回合衰减 2%，并设置最小探索率为 0.05 以保持随机探索。

具体实现如下：

```python
def learn(self, s: int, a: int, r: float, s_: int, done: bool) -> None:
    q_predict = self.q_table[s, a]

    if not done:
        q_target = r + self.gamma * np.max(self.q_table[s_, :])
    else:
        # 终止状态下，目标值为即时奖励
        q_target = r

    self.q_table[s, a] += self.lr * (q_target - q_predict)
```

```python
def predict(self, state: int) -> int:
    """训练模式：ε-贪心算法用于动作选择"""
    if np.random.uniform() > self.epsilon:
        # 利用：选择 Q 值最大的动作
        # 引入随机扰动打破相同值的僵局
        state_action = self.algorithm.q_table[state, :]
        # np.where 返回最大值的索引列表，随机选一个
        action = np.random.choice(np.where(state_action == np.max(state_action))[0])
    else:
        # 探索：随机选择
        action = np.random.choice(self.n_actions)
    return action
```

图 1: 上：Q-value 更新实现；下：ϵ-greedy 行为策略实现

2.3 训练结果与可视化

对模型训练 100 个 episode，结果如下：

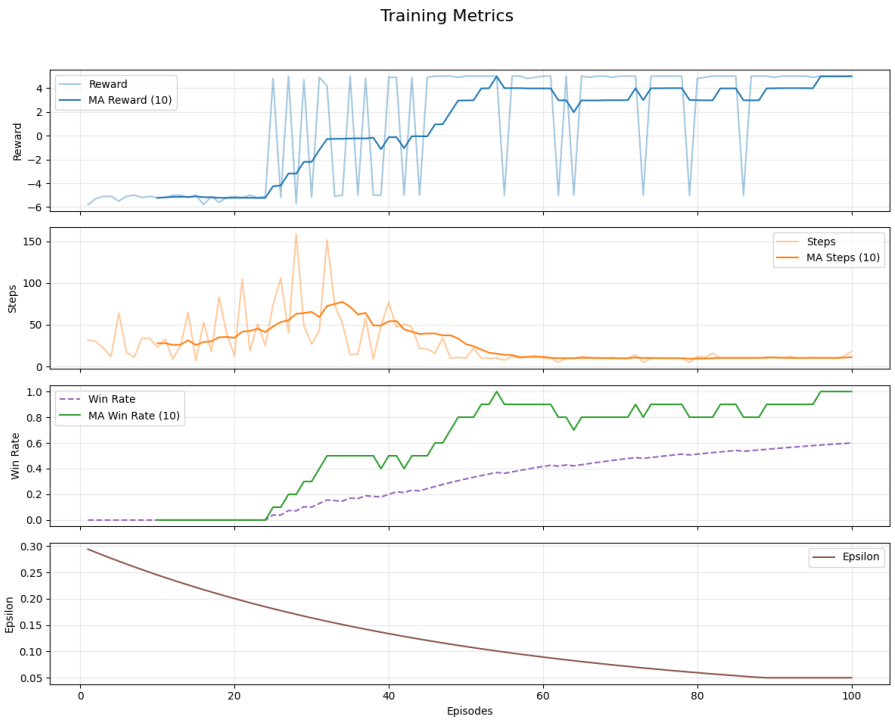

图 2: Q-learning 训练曲线

从上到下分别为：每回合与滑动平均奖励、每回合与滑动平均步数、总体与滑动平均胜率、每回合 ϵ。随着训练进行，平均奖励逐渐提升，步数逐渐减少，表明策略在不断优化。

训练过程中路径变化如下：

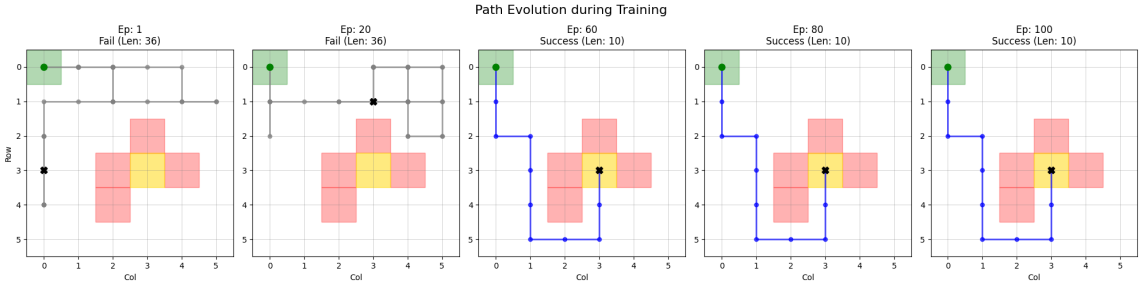

图 3: Q-learning 智能体路径变化

表明策略在约 60 回合后趋于稳定并得到了最佳路径之一。最终的 Q-table 如下：

State	Up	Down	Right	Left
"(0,0)"	-0.052170310000000004	0.1202852097270759	0.0	-0.052160258378064706
"(0,1)"	-0.052170310000000004	5.098915267377607e-05	0.0	0.0
"(0,2)"	-0.052170310000000004	0.0	0.0	0.0
"(0,3)"	-0.052170310000000004	0.0	4.089438495000002e-09	0.0
"(0,4)"	-0.040951	0.0	4.5412377467400034e-07	0.0
"(0,5)"	-0.034390000000000004	2.4571487122308015e-05	-0.034390000000000004	0.0
"(1,0)"	0.001234449705991967	0.2616846609325103	4.34820299136202e-07	-0.06926163252587048
"(1,1)"	0.0	0.1736231258958127	0.0	0.0
"(1,2)"	0.0	0.0	0.0	0.009201033129767363
"(1,3)"	0.0	-2.8476639500000003	0.0	0.0
"(1,4)"	0.0	0.0	0.0	0.0
"(1,5)"	0.0	0.0004260228143847302	-0.019000000000000003	0.0
"(2,0)"	0.003880418879003113	0.0	0.556741752622602	-0.036841712557739166
"(2,1)"	0.002467690128661576	0.9931021105820097	0.0	0.0
"(2,2)"	1.5709900828950007e-09	-0.95	-0.95	0.0
"(2,3)"	0.0	0.0	0.0	0.0
"(2,4)"	0.0	-0.5	0.0	-0.5
"(2,5)"	1.2717128038320007e-05	0.004924981287315002	-0.027100000000000006	0.0
"(3,0)"	0.049099785338343443	0.0	0.0	-0.019000000000000003
"(3,1)"	0.0001569813230859733	1.565388558446883	-3.43094701955	0.0001995654627652373 6
"(3,2)"	0.0	0.0	0.0	0.0
"(3,3)"	0.0	0.0	0.0	0.0
"(3,4)"	0.0	0.0	0.0	0.0
"(3,5)"	0.0	0.04950756212265001	-0.019000000000000003	-0.95
"(4,0)"	0.0	0.0	0.11141061729327276	-0.019000000000000003
"(4,1)"	0.0	2.321250441815494	-2.342795	0.0009507076057953116
"(4,2)"	0.0	0.0	0.0	0.0
"(4,3)"	4.991014948500428	0.0	0.050794573499999995	-0.95
"(4,4)"	-0.95	0.0	0.0	1.564790958672327
"(4,5)"	0.0	0.0	-0.010000000000000002	0.3383042086614196
"(5,0)"	0.0	-0.010000000000000002	0.0001774750500000003	-0.019000000000000003
"(5,1)"	0.0	0.0062574824750685695	3.115015527101628	0.0
"(5,2)"	-1.355	-0.010000000000000002	3.8359163231974636	0.0
"(5,3)"	4.437111872976577	0.7077107263358949	0.0	0.326976204481112
"(5,4)"	0.08345986065	0.0	0.0	0.0
"(5,5)"	0.0	-0.010000000000000002	-0.010000000000000002	0.0018768105000000004

图 4: 最终 Q-table

3 2-step Q-learning 算法

标准 Q-learning 仅利用一步的反馈进行更新。为了让智能体看得更远，实现了 N-step Q-learning，重点分析 $n = 2$ 的情况。

3.1 更新公式与符号含义

N-step Q-learning 的标准更新公式如下：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\sum_{k=1}^{N} \gamma^{k-1} R_{t+k} + \gamma^N \max_a Q(S_{t+N}, a) - Q(S_t, A_t) \right] \tag{7}$$

对于 $n = 2$，具体为：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 \max_a Q(S_{t+2}, a) - Q(S_t, A_t) \right] \tag{8}$$

其中：

- S_t, A_t：时刻 t 的状态和采取的动作。
- $Q(S_t, A_t)$：状态-动作对 (S_t, A_t) 的 Q 值，表示在状态 S_t 下执行动作 A_t 的长期回报估计。
- R_{t+1}：在时刻 t 执行 A_t 后获得的即时奖励。
- R_{t+2}：在时刻 $t+1$ 采取动作 A_{t+1} 后获得的奖励，其中 A_{t+1} 由行为策略 π 产生。
- S_{t+2}：时刻 $t+2$ 的状态，即在 S_{t+1} 下执行 A_{t+1} 后转移到的状态。
- $\max_a Q(S_{t+2}, a)$：对两步之后状态价值的估计：状态 S_{t+2} 下对所有可能动作 a 的 Q 值取最大。
- γ^2：由于向后看了两步，未来的估计值需要进行两次折现。

3.2 更新时机

问题：在什么时候更新 t 时刻对应的参数 $Q(S_t, A_t)$?

回答：在时刻 $t+2$ 发生后。当智能体执行完动作 A_{t+1} 后并获得奖励 R_{t+2} 和状态 S_{t+2}，此时已收集到更新所需所有信息 $(R_{t+1}, R_{t+2}, S_{t+2})$，即可回溯更新 t 时刻的 $Q(s_t, a_t)$；或者在 episode 结束时，对剩余 transition 进行无估计的回报更新。

3.3 代码实现

通过 `collections.deque` 实现长度为 N 的经验缓冲区，用于存储最近的 N 步 transition。当缓冲区满 N 步或 episode 结束时，进行更新，具体实现如下：

```python
def learn(self, s: int, a: int, r: float, s_: int, done: bool) -> None:
    # 1. 存 transition
    self.memory.append((s, a, r, s_))

    # 2. buffer 填满 N 步或遇到终止状态才更新
    if len(self.memory) < self.n_step and not done:
        return

    # 3. 当 buffer ≥ n，更新最早的 transition
    if len(self.memory) >= self.n_step:
        # 计算 N-step Return (G)
        # G = r_0 + gamma * r_1 + ... + gamma^(k-1) * r_{k-1} + gamma^k * max_Q(s_final)
        G = 0.0
        for i in range(self.n_step):
            _, _, r_i, _ = self.memory[i]
            G += self._gamma_pow[i] * r_i

        # 加上未来估值: gamma^N * max Q(s_{t+N})
        s_n = self.memory[self.n_step - 1][3]
        G += self._gamma_pow[self.n_step] * np.max(self.q_table[s_n])

        # 取出需要更新的时间步的数据 (Tn)
        s_0, a_0, _, _ = self.memory.popleft()
        # 更新 Q 值
        self.q_table[s_0, a_0] += self.lr * (G - self.q_table[s_0, a_0])

    # 4. episode 结束后 flush buffer
    if done:
        while self.memory:
            G = 0.0
            for i, (_, _, r_i, _) in enumerate(self.memory):
                G += self._gamma_pow[i] * r_i

            s_0, a_0, _, _ = self.memory.popleft()
            self.q_table[s_0, a_0] += self.lr * (G - self.q_table[s_0, a_0])
```

图 5: 2-step Q-learning 更新实现

3.4 训练结果与可视化

相关超参数与 Q-learning 相同，对模型训练 100 个 episode，结果如下：

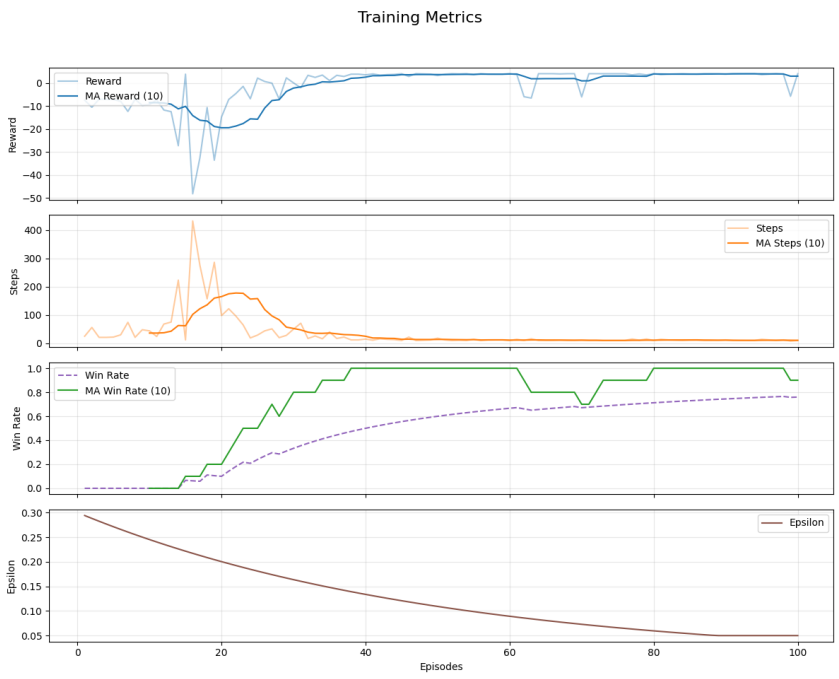

图 6: 2-step Q-learning 训练曲线

对比 Q-learning，2-step 方法在平均奖励和步数上均表现出更快的收敛速度，表明其利用多步回报信息加速了学习过程。训练过程中路径变化如下：

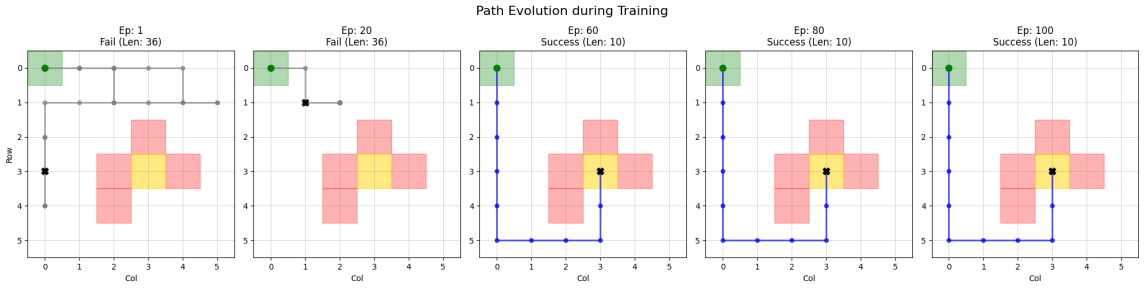

图 7: Q-learning 智能体路径变化

表明策略在约 60 回合后趋于稳定并得到了最佳路径之一。注意到 2-step 方法得出了与标准 Q-learning 不同的另一种最佳路径，这体现了多步方法在探索策略空间时的差异性，也与随机性等因素有关。

最终的 Q-table 如下：

State	Up	Down	Right	Left
"(0,0)"	-0.5795439561371682	0.7711511508489782	-0.49404926033595187	-0.5474711985859566
"(0,1)"	-0.5581064200209381	-0.5527108087356489	-0.5546307673701669	-0.4024817094424001
"(0,2)"	-0.5190906471619715	-0.5163529343590593	-0.5107977758721329	-0.4862283854946508
"(0,3)"	-0.5165268474788444	-0.5852775756830408	-0.5093821937068842	-0.5177947671890561
"(0,4)"	-0.4809548668862838	-0.4661300767399868	-0.4679094441201491	-0.47613748722014154
"(0,5)"	-0.4715469185795454	-0.46505047311688263	-0.4682962341283192	-0.4747951919071634
"(1,0)"	-0.44013494910813755	1.0990883918827639	-0.45288030350828173	-0.404521095460191
"(1,1)"	-0.5271688439200343	-0.514841419769289	-0.5206434498589543	-0.3672583861455215
"(1,2)"	-0.550619175247288	-1.0531474312295008	-0.8612811373717811	-0.5487530595253258
"(1,3)"	-0.34796518847587393	-1.7195	-0.3315790347525964	-0.32485263399534214
"(1,4)"	-0.48952482465492486	-0.47949265538974006	-0.47777610833459533	-0.8623728863397582
"(1,5)"	-0.45389399248989903	-0.33998377119511597	-0.43941427251265364	-0.4386214532477212
"(2,0)"	-0.3097378866730739	1.382636079231304	-0.3463550576543149	-0.06436884133748366
"(2,1)"	-0.5571589650091274	-0.7289434133785169	-0.8298135276170646	-0.3496000139728514
"(2,2)"	-0.22103310912150623	-0.95	-1.355	-0.2410359662540735
"(2,3)"	0.0	0.0	0.0	0.0
"(2,4)"	-0.18671208833090827	-0.5	-0.1763135154388197	-0.95
"(2,5)"	-0.46063215906804245	-0.27110320853949993	-0.4533768880068442	-0.7804030460477849
"(3,0)"	-0.4516950888255934	1.7655712484461872	-0.2870980691120232	-0.4560656581327684
"(3,1)"	-0.3787032741328848	-0.4459584999270517	-1.355	-0.15408676626655404
"(3,2)"	0.0	0.0	0.0	0.0
"(3,3)"	0.0	0.0	0.0	0.0
"(3,4)"	0.0	0.0	0.0	0.0
"(3,5)"	-0.037639000000000006	0.15853992200000006	-0.08307672152500001	-0.5
"(4,0)"	-0.3406547710469756	2.077335625821041	-0.10455870791699778	0.008854020964336573
"(4,1)"	-0.44950836960802637	-0.28166892080822215	-1.355	0.34520979791136586
"(4,2)"	0.0	0.0	0.0	0.0
"(4,3)"	4.9983350518173415	0.5702067585758589	0.31845741411635037	-1.355
"(4,4)"	0.0	0.0	0.126949364	2.6953498484000002
"(4,5)"	0.0	-0.01856985445	0.0	0.5962593907016451
"(5,0)"	-0.40751167747887024	-0.18661963758045214	2.688965806401269	0.22917388586948384
"(5,1)"	-0.4772644285858536	-0.11656962010838293	2.4053644142009745	-0.04170949175383032
"(5,2)"	-0.95	0.7814130602882292	3.8278626757313394	-0.04393893722506585
"(5,3)"	4.308466452761624	0.46509194034305	0.3342229727120001	0.2825144990282106
"(5,4)"	0.7818492711848718	0.031616000000000005	0.0	0.0
"(5,5)"	0.10356596000000003	-0.019000000000000003	-0.019000000000000003	-0.019000000000000003

图 8: 最终 Q-table

4 分析与思考

在完成基本的算法实现后，针对不同参数设置和环境配置进行了对比实验，观察到两个值得探讨的现象。

4.1 固定探索率下 2-step 方法的震荡问题

实验观察：在实验中发现，当采用 $n = 2$ 的 Q-learning 且探索率 ϵ 固定为 0.1 时，策略容易在迷宫的某些局部状态间形成往复行为，导致无法收敛到最优路径。相比之下，同样设置下的标准 Q-learning 表现较为稳定。

原因分析：该现象源于 N-step off-policy 方法中 Bootstrap 所引入的**最大化偏差**传播。

1. 偏差放大：在 $Q(S_{t+2}, a)$ 的估计中使用了 max 操作，这本身会带来高估。在 $n = 2$ 的情况下，这种高估值易被更快地反向传播。
2. 价值抬高：由于高估，相邻状态的动作价值被相互抬高。
3. 极限环：在固定 $\epsilon = 0.1$ 的探索扰动下，智能体总是有一定概率偏离当前最优策略。在 $n = 2$ 时，这种偏离带来的反馈（可能是错误的路径）可能会因为视野更长而更强烈地影响前序节点，导致策略无法稳定，形成从 $S_A \to S_B \to S_A$ 的稳定极限环。

解决方案：

- ϵ **衰减**：引入 ϵ 衰减机制，随着训练进行减少随机扰动，迫使策略稳定。
- **Double Q-learning**：引入 Double Q-learning：

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 Q_2(S_{t+2}, \arg\max_a Q_1(S_{t+2}, a)) - Q_1(S_t, A_t) \right]$$

维护两个 Q 表并交替更新 Q_1 和 Q_2，解耦动作选择和价值评估：

```python
if np.random.rand() < 0.5:
    # Use Q1 to select action, Q2 to evaluate
    a_star = np.argmax(self.q1[s_n])
    G += self._gamma_pow[self.n_step] * self.q2[s_n, a_star]
    s0, a0, _, _ = self.memory.popleft()
    self.q1[s0, a0] += self.lr * (G - self.q1[s0, a0])
else:
    # Use Q2 to select action, Q1 to evaluate
    a_star = np.argmax(self.q2[s_n])
    G += self._gamma_pow[self.n_step] * self.q1[s_n, a_star]
    s0, a0, _, _ = self.memory.popleft()
    self.q2[s0, a0] += self.lr * (G - self.q2[s0, a0])
```

图 9: Double Q-learning 代码实现

实验证明，在 $N = 2$ 且 ϵ 固定的情况下，Double Q 能有效消除最大化偏差，破坏震荡的形成条件。

4.2 步数惩罚下 N-step 的优势

实验观察：在引入每步 -0.1 的移动惩罚（即 $R_{\text{step}} = -0.1$）后，实验发现 $N = 2$ 的 Q-learning 能够更快收敛并找到最短路径，而 $N = 1$ 的标准 Q-learning 反而收敛极慢，甚至陷入局部最优。

原因分析：

1. 问题性质转变：引入 Step Penalty 后，问题从稀疏奖励 MDP 转变为密集奖励 MDP，此时路径长度成为了显式的优化目标。

2. 信息累积差异：

   - 标准 Q-learning：每次更新仅能看到当前的 -0.1 和对未来的估计。由于初始 Q 值为 0，负奖励的反向传播缓慢，梯度平缓，策略难以区分长期成本上的微小差异；且更容易受 ϵ 噪声干扰，导致策略振荡。
   - 2-step Q-learning：一次更新包含 $R_{t+1} + \gamma R_{t+2}$。这意味着算法在单次更新中就能累积多步的负奖励。对于长路径，负反馈积累得更快、更显著。

3. 区分能力：N-step 方法能够更快地拉大短路径与长路径之间的价值差异。

结论：在存在持续步长惩罚或密集负反馈的环境中，多步 TD 方法（如 2-step Q-learning）相比一步方法具有更强的价值辨识能力和收敛稳定性。

5 总结

本项目实现了基于强化学习 Q-learning 算法和两步 Q-learning 算法解决了迷宫宝藏问题，找到其中几条最佳路径。

通过对比实验，验证了 N-step Q-learning 在处理稀疏奖励和长程依赖问题上的理论优势，同时也展示了其在 Off-policy 设置下对超参数更为敏感的特性。结合 Double Q-learning 和合理的奖励塑形，可以显著提升算法的鲁棒性。

在代码实践中，我深化了强化学习的基础理论，也获得了以下体会：

1. 理论与实践的差异：课堂上贝尔曼方程在实际编码中需要精细处理边界条件和数据结构等细节，它决定算法能否收敛。

2. 超参数：ϵ-greedy 策略的衰减曲线设计对收敛速度和性能影响较大。过快的衰减会导致陷入局部最优，而过慢的衰减则浪费训练时间，这正是探索与利用这一矛盾的权衡。

3. 对 TD 方法的深入理解：通过对比 1-step 和 2-step 算法在不同环境设置下的表现，我直观地理解了偏差 Bias 与方差 Variance 的权衡。多步更新虽然能加快奖励传播，但也更容易引入累积误差。这为后续使用更复杂的 RL 算法如 DQN 打下坚实基础。