華中科技大學

研究生课程报告

姓　　名	张昭骏
学　　号	M202573989
系、年级	计算机科学与技术 25 级
类　　别	全日制
报告科目	强化学习
日　　期	2025 年 10 月 25 日

目 录

1 引言

在本次课程实验中，我们小组选择围绕《吃豆人》（Pacman）游戏环境，探索和实现了多种强化学习算法。其中我负责训练环境的搭建，并实现了 Monte Carlo Learning（MC Learning）、Q-Learning 以及 Approximate Q-Learning 三种基于值函数的强化学习方法。

1.1 Pacman 游戏

在 Pacman 游戏中，玩家控制吃豆人在迷宫中上下左右移动，地图中存在固定数量的食物（food）和胶囊（capsule），以及随机移动的鬼（ghost）。当吃豆人吃掉所有食物时，游戏胜利。

Pacman 作为经典游戏，其状态空间大小适中，奖励结构清晰，既不过于简单也不过于复杂，适合算法的实现与测试；环境兼具确定性与随机性，鬼的随机移动增加了不确定性，考验算法的鲁棒性；此外，游戏具有长期规划特性，agent 需要在避开鬼和收集食物之间进行权衡，胶囊机制也增加了策略的多样性。

在本次实验中，我们选择 Pacman 作为强化学习实验的环境，探索不同算法在该环境下的表现与效果。

1.2 基于值的强化学习方法

强化学习方法可以大致分为基于值的方法（value-based）、基于策略的方法（policy-based）以及结合两者的混合方法（actor-critic）。基于值的强化学习方法通过估计一个值函数（如状态值函数 $V(s)$ 或动作值函数 $Q(s,a)$）来指导智能体的决策，是一类经典的 model-free 强化学习方法。

在本次实验中，我依次实现了 MC learning、Q-Learning 以及 Approximate Q-Learning 三种算法。

MC learning 与 Q-Learning 算法是相似的，两者都通过学习动作值函数 $Q(s,a)$ 来指导决策，区别主要在于值函数的更新方式：MC learning 基于完整回合的实际回报进行更新，而 Q-Learning 则采用时序差分（Temporal Difference, TD）方法，每一步都可以进行增量式更新，学习效率更高。通过实现这两种相

对简单的方法，我加深了对强化学习基本原理和学习过程的理解。

然而，将 $Q(s,a)$ 直接作为学习目标，面临着状态空间爆炸的问题。因此，我进一步实现了 Approximate Q-Learning 算法，通过手工设计的特征函数将高维状态映射到低维特征空间，再使用线性函数逼近来估计 Q 值，从而大大降低了状态空间的复杂度，使算法能够泛化到未见过的状态，并在更复杂的环境中取得良好的表现。

1.3 报告结构

本文的组织结构如下：

第 2 章介绍三种强化学习方法的原理与设计，包括问题的 MDP 建模、MC learning、Q-Learning 以及 Approximate Q-Learning 的基本原理和算法设计；

第 3 章详细阐述各算法的具体实现，包括强化学习训练环境的搭建、各算法的代码实现细节；

第 4 章展示实验结果与分析，对比三种算法在不同地图规模下的性能表现；

第 5 章对本次实验进行总结，并提出未来可能的改进方向。

2 方法原理与设计

本章首先结合 Pacman 游戏规则进行问题建模，之后逐一介绍各算法的原理和设计。

2.1 问题建模

在将强化学习算法应用于 Pacman 游戏之前，我们首先需要明确游戏的运行机制，并将问题建模为马尔可夫决策过程（Markov Decision Process, MDP）。

2.1.1 Pacman 游戏环境

本实验采用的 Pacman 游戏环境基于开源 Pacman 项目改编，并封装为符合 Gymnasium 标准的强化学习环境。

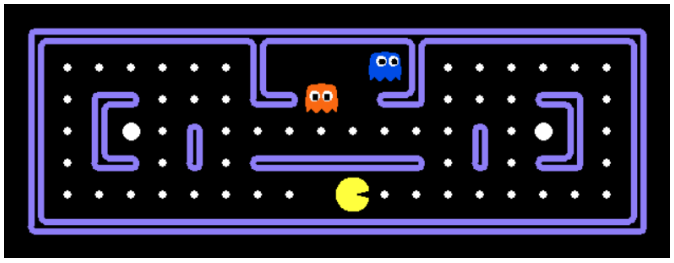

图 1 Pacman 游戏画面示例

游戏的基本要素包括：地图（Layout）由墙壁（wall）、通道、食物（food）、胶囊（capsule）以及角色初始位置组成，以文本文件形式存储，如图2所示。吃豆人由智能体控制，可以在迷宫中上下左右移动（North, South, East, West），也可以选择停留（Stop），其位置用坐标 (x, y) 表示。食物和胶囊分布在地图的通道中，吃豆人吃掉食物可以获得分数，吃掉胶囊则会使所有鬼进入恐慌状态。

layouts > smallClassic.lay

```
1  %%%%%%%%%%%%%%%%%%%%
2  %......%G  G%......%
3  %.%%...%%  %%...%%.%
4  %.%o.%........%.o%.%
5  %.%%.%.%%%%%.%.%%.%
6  %........P.........%
7  %%%%%%%%%%%%%%%%%%%%
8
```

图 2 地图文本表示

鬼（Ghost）是随机移动的敌对角色，通常采用随机策略选择动作，有正常状态和恐慌状态两种：正常状态下碰到吃豆人会导致游戏失败；而在恐慌状态下（吃豆人吃掉胶囊后的 40 个时间步内），被吃豆人吃掉后会在初始位置复活并解除恐慌。

游戏的终止条件包括：吃豆人吃掉所有食物则游戏胜利；吃豆人碰到处于正常状态的鬼则游戏失败；或达到最大步数限制（通常设置为 1000 步）。

2.1.2 马尔可夫决策过程建模

强化学习算法的核心是通过与环境交互来学习最优策略，而 MDP 通过定义状态、动作、转移概率和奖励函数，将这一交互过程形式化为值函数的优化问题。MDP 的马尔可夫性质（下一状态仅依赖于当前状态和动作）使得贝尔曼方程成立，从而可以递归地求解最优值函数和策略。以下将 Pacman 游戏的各个要素形式化为 MDP 的组成部分。

状态空间 $\mathcal{S}$：游戏中的完整信息构成状态。具体地，状态包括吃豆人的位置 (x_p, y_p) 和移动方向，所有鬼的位置 (x_g^i, y_g^i) 和恐慌计时器 t_{scared}^i（$i = 1, 2, \ldots, n_g$），剩余食物的分布（用布尔矩阵 F 表示，$F[x][y] = 1$ 表示位置 (x, y) 有食物），剩余胶囊的位置集合 C，以及当前累积分数 s。这些信息描述了游戏的当前局面，并满足马尔可夫性质：给定当前状态和动作，下一状态的分布与历史无关。

值得注意的是，状态空间的大小随地图尺寸呈指数级增长。以 `smallClassic` 地图（尺寸为 20×7）为例，假设有 2 个鬼和 30 个食物点，完整状态空间的规模约为 $|\mathcal{S}| \approx 140 \times 140^2 \times 40^2 \times 2^{30} \approx 10^{17}$，庞大的状态空间给强化学习算法的设计和实现带来了挑战。

动作空间 $\mathcal{A}$：智能体在每个时间步可以选择五个基本动作之一，即 $\mathcal{A} = \{\text{North}, \text{South}, \text{East}, \text{West}, \text{Stop}\}$。在选择动作时还需考虑墙壁约束，只有合法动作（不会撞墙的动作）可被选择。

状态转移函数 $P(s'|s, a)$：游戏规则决定了状态如何转移，这部分由开源 Pacman 项目代码实现。给定当前状态 s 和吃豆人的动作 a，状态转移包括确定性和随机性两部分：吃豆人根据动作 a 确定性地移动到新位置，食物和胶囊在被吃掉时确定性地更新，恐慌计时器确定性地递减；而鬼根据其随机策略选择动

作并移动，引入了转移的随机性。因此，虽然吃豆人的行为是确定的，但由于鬼的随机移动，整体状态转移 $P(s'|s,a)$ 是一个概率分布。

奖励函数 $R(s,a,s')$：游戏的计分规则被形式化为奖励函数，以引导智能体的学习。具体地，吃掉一个食物获得 $+10$ 分，吃掉一个恐慌状态的鬼获得 $+200$ 分，吃掉所有食物（胜利）额外获得 $+500$ 分，碰到正常状态的鬼（失败）扣除 500 分，每走一步扣除 1 分作为时间惩罚。吃掉胶囊本身不直接给分，但会触发鬼进入恐慌状态，间接影响后续奖励。

折扣因子 γ：设置 $\gamma = 0.8$，用于平衡即时奖励和长期奖励。较小的折扣因子使得智能体更重视近期奖励，倾向于尽快吃掉食物并避开鬼，而不是过度探索。

通过以上建模，Pacman 游戏被形式化为一个 MDP 五元组 $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$。智能体的目标是学习最优策略 π^*，最大化从初始状态开始的期望累积折扣奖励：

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \tag{1}$$

2.2 Monte Carlo Learning 原理与设计

2.2.1 基本原理

MC learning 是一类基于采样的强化学习算法，其核心思想是通过完整的 episode 经验来估计值函数。与需要环境模型的动态规划方法不同，MC learning 仅需与环境交互即可学习。

动作值函数 $Q(s,a)$ 定义为从状态 s 执行动作 a 后，遵循策略 π 所能获得的期望累积折扣奖励：

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a\right] \tag{2}$$

MC learning 通过实际经验来估计这一期望值。具体而言，智能体与环境交互产生完整的 episode：

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_T \tag{3}$$

其中 S_T 是终止状态。对于 episode 中出现的每个状态-动作对 (s,a)，可以计算从该时刻开始的实际回报（return）：

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \tag{4}$$

MC learning 的基本思想是用实际回报 G_t 的样本平均来估计 $Q(s,a)$。根据

大数定律，随着采样 episode 数的增加，样本平均将收敛到真实的期望值。

在遍历 episode 的每一步时，可能重复遇到某些状态-动作对，相应地在更新值函数时有两种策略：first-visit 和 every-visit。first-visit 只在 episode 中首次访问 (s, a) 时更新 $Q(s, a)$，而 every-visit 在每次访问时都进行更新。本实验采用 every-visit 策略，配合增量式更新来提高学习效率：

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(G_t - Q(s, a) \right) \tag{5}$$

其中 $\alpha \in (0, 1]$ 是学习率，控制新样本对估计值的影响程度。

此外，为了在探索（exploration）和利用（exploitation）之间平衡，采用 ϵ-Greedy 贪心策略选择动作：以概率 $1 - \epsilon$ 选择当前估计下的最优动作 $\arg\max_a Q(s, a)$，并以概率 ϵ 随机选择动作（作为一种尝试）。

2.2.2 算法设计

基于前述原理，MC learning 的训练过程可以形式化为算法 1。算法采用 every-visit 策略配合增量式更新，通过反复与环境交互来逐步改进 Q 值估计和策略。算法流程如 1 所示，推理时只需将 α 和 ϵ 设为 0 即可。

每个训练轮次包括两个阶段：数据收集阶段（第 5-12 行）和值函数更新阶段（第 14-18 行）。在数据收集阶段，智能体使用 ϵ-Greedy 策略与环境交互，生成完整的 episode 并存储在缓冲区中。该策略在利用当前最优动作（exploitation）和随机探索（exploration）之间取得平衡，确保算法能够充分探索状态空间。

episode 结束后进入更新阶段，算法从后向前遍历缓冲区中的经验。对于每个时间步 t，首先根据递推关系 $G_t = R_{t+1} + \gamma G_{t+1}$ 计算该步的累积折扣回报 G_t（其中 $G_T = 0$），然后使用式 (5) 更新对应状态-动作对的 Q 值。这一增量式更新方式使得 Q 值逐渐向真实值函数靠拢，同时学习率 α 控制了新样本对估计的影响程度。

对于 Q 函数，这里实现为字典（dictionary）结构，以完整游戏状态为键，存储对应的动作值。状态与 MDP 建模部分定义相同，由于 Pacman 游戏的状态空间可能很大，字典仅存储实际访问过的状态-动作对，未访问的对其 Q 值默认为 0。

经过足够多 episode 的训练，算法将收敛到最优值函数 Q^*，此时贪心策略

算法 1 Monte Carlo Learning

Input: 环境 env，最大训练 episode 数 N

Output: 学习得到的 Q 值函数 Q

初始化：$Q(s,a) \leftarrow 0$ 对所有 $s \in \mathcal{S}, a \in \mathcal{A}$
设置超参数：$\alpha = 0.1, \epsilon = 0.05, \gamma = 0.8$
for $episode = 1$ to N **do**
    $s \leftarrow$ env.reset()　　▷ 初始化环境
    buffer $\leftarrow []$　　▷ 初始化 *episode* 缓冲区
    while episode 未结束 **do**
        以 ϵ-Greedy 策略选择动作：
        $$a \leftarrow \begin{cases} \arg\max_{a'} Q(s,a') & \text{概率} 1-\epsilon \\ \text{random}(\mathcal{A}) & \text{概率} \epsilon \end{cases}$$
        $s', r \leftarrow$ env.step(a)　　▷ 执行动作并观察转移
        buffer.append$((s,a,r))$　　▷ 存储经验
        $s \leftarrow s'$
    end while
    $G \leftarrow 0$　　▷ 从终止状态开始反向计算回报
    for $t = |\text{buffer}| - 1$ down to 0 **do**
        $(s_t, a_t, r_{t+1}) \leftarrow$ buffer$[t]$
        $G \leftarrow r_{t+1} + \gamma G$　　▷ 累积折扣回报
        $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(G - Q(s_t,a_t))$　　▷ 更新 Q 值
    end for
end for
return Q

$\pi(s) = \arg\max_a Q(s,a)$ 即为最优策略。

2.3 Q-Learning 原理与设计

2.3.1 基本原理

Q-Learning 是一种时序差分（Temporal Difference, TD）学习算法，与 MC learning 的主要区别在于值函数的更新时机和方式。MC learning 需要等待完整 episode 结束后才能计算回报 G_t 并更新 Q 值，而 Q-Learning 在每一步转移后即可进行更新，无需等待终止状态。

Q-Learning 的核心更新规则基于贝尔曼最优方程，使用单步转移的即时奖

励和下一状态的最优值估计来逼近真实 Q 值：

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{6}$$

其中 (s,a,r,s') 是观察到的一步转移。与 MC learning 使用完整回报 G_t 不同，Q-Learning 使用 TD 目标 $r + \gamma \max_{a'} Q(s',a')$ 作为 Q 值的估计。这一目标结合了即时奖励 r 和下一状态的最优值估计 $\max_{a'} Q(s',a')$，形成了自举（bootstrapping）式的更新。

2.3.2 算法设计

Q-Learning 的训练过程如算法 2 所示。相比 MC learning，该算法无需维护 episode 缓冲区，每步转移后立即更新 Q 值。

算法 2 Q-Learning 算法

Input: 环境 env，最大训练 episode 数 N
Output: 学习得到的 Q 值函数 Q

1: 初始化：$Q(s,a) \leftarrow 0$ 对所有 $s \in \mathcal{S}, a \in \mathcal{A}$
2: 设置超参数：$\alpha = 0.1, \epsilon = 0.05, \gamma = 0.8$
3: **for** $episode = 1$ to N **do**
4: $s \leftarrow$ env.reset()
5: **while** episode 未结束 **do**
6: 以 ϵ-Greedy 策略选择动作：
7: $$a \leftarrow \begin{cases} \arg\max_{a'} Q(s,a') & \text{概率} 1-\epsilon \\ \text{random}(\mathcal{A}) & \text{概率} \epsilon \end{cases}$$
8: $s', r \leftarrow$ env.step(a)
9: $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$ ▷ *TD 更新*
10: $s \leftarrow s'$
11: **end while**
12: **end for**
return Q

算法在每次状态转移后（第 10 行）立即执行 TD 更新。更新目标 $r + \gamma \max_{a'} Q(s',a')$ 结合了当前步的即时奖励和对未来回报的估计，这种单步前瞻（one-step lookahead）的方式使得算法能够快速传播价值信息，加速学习过程。Q 函数同样实现为字典结构，存储方式与 MC learning 完全相同。

2.4 Approximate Q-Learning 原理与设计

前述两种方法（MC learning 和 Q-Learning）在实现 Q 函数时，都是直接记录每个状态-动作对的值，可以视作在填一张表格：当表格填满时，意味着所有状态-动作对都被访问过，从而完整得到 Q 函数。智能体在决策时，通过查表得到当前状态下的最优 Q 值，进而做出最佳的动作。然而，这同时也意味着如果某个状态-动作对从未被访问过，其 Q 值就为初始值，即使见过与其相似的状态-动作对，智能体也无法做出合理的决策；此外，如前所述，Pacman 游戏的状态空间随地图尺寸呈指数级增长，当地图稍大时，Q 函数就变得难以学习。

Approximate Q-Learning 方法通过引入函数近似技术和状态特征设计，解决了上述问题。

2.4.1 函数近似

函数近似（function approximation）通过参数化方式表示 Q 函数，将 Q 函数从离散的查表转换为连续函数逼近。其核心思想是将 Q 值表示为特征的线性组合：

$$Q(s,a) = \sum_{i=1}^{n} w_i f_i(s,a) = \mathbf{w}^\top \mathbf{f}(s,a) \tag{7}$$

其中 $\mathbf{f}(s,a) = [f_1(s,a), f_2(s,a), \ldots, f_n(s,a)]^\top$ 是从状态-动作对提取的特征向量，$\mathbf{w} = [w_1, w_2, \ldots, w_n]^\top$ 是对应的权重向量。特征函数 $f_i(s,a)$ 将高维状态空间映射到低维特征空间，捕获对决策有用的关键信息，而权重 w_i 则刻画了各特征对 Q 值的贡献程度。

在这一表示下，Q-Learning 的更新规则转化为对权重向量的更新。给定一步转移 (s,a,r,s')，TD 误差（temporal difference error）定义为

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a) \tag{8}$$

该误差衡量了当前 Q 值估计与 TD 目标之间的差距。利用梯度下降法最小化 TD 误差的平方，可得权重更新规则：

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta \cdot \mathbf{f}(s,a) \tag{9}$$

该更新规则沿着减小 TD 误差的方向调整权重，其中学习率 α 控制更新步长。

相比表格式方法，函数近似具有显著优势。首先，参数量大幅减少，从 $|\mathcal{S}| \times |\mathcal{A}|$

降至特征维度 n（通常 $n \ll |\mathcal{S}| \times |\mathcal{A}|$）。更重要的是，函数近似实现了泛化能力：类似于插值，当两个不同的状态-动作对 (s_1, a_1) 和 (s_2, a_2) 提取出相似的特征，即 $\mathbf{f}(s_1, a_1) \approx \mathbf{f}(s_2, a_2)$ 时，它们将得到相似的 Q 值估计 $\mathbf{w}^\top \mathbf{f}(s_1, a_1) \approx \mathbf{w}^\top \mathbf{f}(s_2, a_2)$。这意味着智能体处理未见过的新状态时，可以利用隐含在权重中的知识做出合理的决策。因此，通过线性逼近，可以对不完全匹配已知的 (s, a) 估计出合理的值，从而不需要在训练中" 完全覆盖表格"。从另一个角度看，也可以认为学习效率提高了，一次权重更新会影响所有具有相似特征的状态-动作对。

2.4.2 特征设计

特征设计是函数近似方法的核心，直接决定了算法的性能上限。特征应当捕获与决策相关的关键信息，同时保持维度适中以确保学习效率。

本实验设计的特征如表 1 所示，共 10 维（包含偏置项）。

表 1 Approximate Q-Learning 特征设计

特征名称	含义	取值范围
`bias`	偏置项	1.0
`#-of-normal-ghosts-1-step-away`	1 步内的普通 ghost 数量	整数
`#-of-scared-ghosts-1-step-away`	1 步内的恐惧 ghost 数量	整数
`eats-scared-ghost`	能否 1 步吃掉恐惧 ghost	$\{0, 1\}$
`closest-scared-ghost`	最近恐惧 ghost 距离	$[0, 1]$
`scared-timer`	恐惧状态剩余时间	$[0, 1]$
`eats-capsule`	能否 1 步吃到胶囊	$\{0, 1\}$
`closest-capsule`	最近胶囊距离	$[0, 1]$
`eats-food`	能否 1 步吃到食物	$\{0, 1\}$
`closest-food`	最近食物距离	$[0, 1]$

距离类特征（`closest-*`）通过除以地图宽度与高度的乘积归一化到 $[0, 1]$ 区间，`scared-timer` 通过除以恐惧状态持续时间（40 个时间步）归一化。这些特征涵盖了 Pacman 游戏中与决策相关的主要信息：食物和胶囊的位置与可达性、普通 ghost 的威胁、恐惧 ghost 的捕获机会、以及恐惧状态的时间窗口。

2.4.3 算法设计

基于线性函数近似和上述特征设计，Approximate Q-Learning 的训练过程如算法 3 所示。

算法 3 Approximate Q-Learning 算法

Input: 环境 env，特征提取器 ϕ，最大训练 episode 数 N
Output: 学习得到的权重向量 $\mathbf{w}$

1: 初始化：$\mathbf{w} \leftarrow \mathbf{0}$（所有权重初始化为 0）
2: 设置超参数：$\alpha = 0.1, \epsilon = 0.05, \gamma = 0.8$
3: **for** $episode = 1$ to N **do**
4: $s \leftarrow$ env.reset()
5: **while** episode 未结束 **do**
6: 以 ϵ-Greedy 策略选择动作：
7: $$a \leftarrow \begin{cases} \arg\max_{a'} \mathbf{w}^\top \phi(s, a') & \text{概率} 1-\epsilon \\ \text{random}(\mathcal{A}) & \text{概率} \epsilon \end{cases}$$
8: $s', r \leftarrow$ env.step(a)
9: $\delta \leftarrow r + \gamma \max_{a'} \mathbf{w}^\top \phi(s', a') - \mathbf{w}^\top \phi(s, a)$ ▷ 计算 *TD* 误差
10: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \cdot \delta \cdot \phi(s, a)$ ▷ 更新权重
11: $s \leftarrow s'$
12: **end while**
13: **end for**
return w

算法的核心在于第 10-11 行的权重更新机制。首先计算 TD 误差 δ（第 10 行），其衡量了当前估计 $\mathbf{w}^\top \phi(s, a)$ 与 TD 目标 $r + \gamma \max_{a'} \mathbf{w}^\top \phi(s', a')$ 之间的差距。随后沿着特征向量 $\phi(s, a)$ 的方向更新权重（第 11 行），更新幅度由 TD 误差 δ 和学习率 α 共同决定。

直观地理解，若某状态-动作对的实际回报高于当前估计（$\delta > 0$），则增大与该对相关的特征权重；反之若实际回报低于估计（$\delta < 0$），则减小相应权重。通过这种方式，算法逐步调整权重向量，使得 Q 值估计 $\mathbf{w}^\top \phi(s, a)$ 逼近真实的最优 Q 函数。

通过特征提取，将大量原始状态映射到少量特征组合，减小了“表格”的大小，从而易于学习；再结合线性逼近带来的泛化性能，Approximate Q-Learning 能够在大规模状态空间中有效完成学习。

3 方法实现

本章介绍三种算法的具体实现细节，包括数据结构设计、核心函数实现以及关键技术选择。算法使用 Python 实现，基于 Gymnasium 标准接口与 Pacman 环境交互。

3.1 环境实现

Pacman 环境基于 UC Berkeley CS188 的 Pacman 项目改编，更新为 Python3 实现并封装为符合 Gymnasium 标准接口的强化学习环境。环境类 `PacmanEnv` 继承自 `gymnasium.Env`，实现了标准的 `reset()`, `step(action)` 等方法。

动作空间定义为离散空间 `spaces.Discrete(5)`，对应五个基本动作（North, South, East, West, Stop）。观测空间使用 `spaces.Dict` 定义，包含吃豆人位置、鬼的位置和状态、食物分布、胶囊位置等多个字段，具体维度在 `reset()` 时根据地图尺寸动态调整。

训练过程中，每个 episode 开始时，调用 `env.reset()` 初始化环境并获取初始状态。在每个时间步，智能体根据当前状态选择动作，通过 `env.step(action)` 执行，环境返回包含下一状态、奖励、终止标志等信息的元组。智能体利用这些信息更新策略，重复上述过程直至 episode 结束。

环境内部维护游戏状态对象 `game.state`，封装了完整的游戏信息。状态对象提供了若干查询方法：`getPacmanPosition()` 返回吃豆人位置坐标，`getGhostStates()` 返回所有鬼的状态列表（包括位置和 `scaredTimer`），`getFood()` 返回食物分布的布尔矩阵，`getCapsules()` 返回胶囊位置列表，`getScore()` 返回当前累积分数。这些方法为算法的决策和特征提取提供了必要的信息。

奖励信号通过相邻两步的分数差计算：`reward = current_score - previous_score`。

3.2 Monte Carlo Learning 实现

MC learning 的实现围绕 Q 表和 episode 缓冲区两个核心数据结构展开。

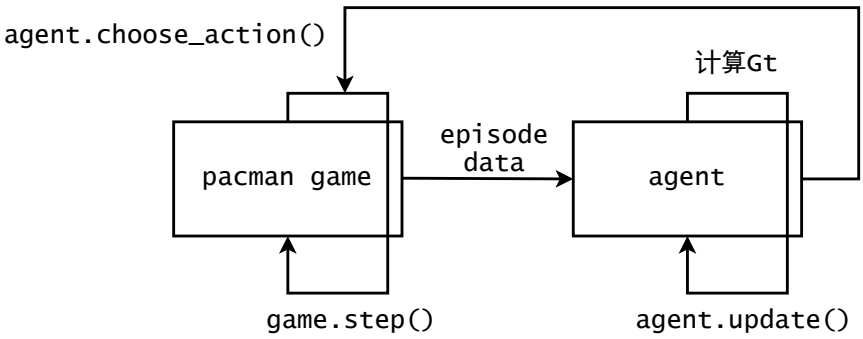

图 3　MC learning 流程图

Q 表使用 Python 字典实现:`Q_values: dict[(state, action)] -> float`。字典的键为状态-动作对元组，值为对应的 Q 值。采用字典而非多维数组的原因在于状态空间极其庞大，字典的稀疏存储可避免预先分配整个空间。未访问过的状态-动作对通过 `get()` 方法默认返回 0.0。

Episode 缓冲区实现为 Python 列表：`episode_buffer: list[(state, action, reward)]`。智能体在每次状态转移后，将当前经验 `(state, action, reward)` 追加到缓冲区。当 episode 结束时，触发 `update_from_episode()` 方法进行批量更新。

回报的反向计算采用递推方式实现。算法从缓冲区末尾开始反向遍历，维护累积回报 G，初始化为 0。对于时间步 t，根据 $G_t = R_{t+1} + \gamma G_{t+1}$ 更新 G 值，然后使用式 (5) 更新对应的 Q 值：

```
G = 0.0
for t in range(len(episode_buffer) - 1, -1, -1):
    state, action, reward = episode_buffer[t]
    G = discount * G + reward


    old_q = Q_values.get((state, action), 0.0)
    Q_values[(state, action)] = old_q + alpha * (G - old_q)
```

这一实现对应算法 1 的第 14-18 行，通过单次反向遍历完成所有 Q 值的更新。

ϵ-Greedy 策略在 `choose_action()` 方法中实现。算法以概率 ϵ 随机选择动作，以概率 $1 - \epsilon$ 选择当前 Q 值最大的动作。在测试阶段，设置 $\epsilon = 0$ 以采用纯贪心策略。

3.3 Q-Learning 实现

Q-Learning 的 Q 表结构与 MC learning 完全相同，均为字典存储。两者的核心区别在于更新时机：Q-Learning 在每次状态转移后立即更新，无需等待 episode 结束。

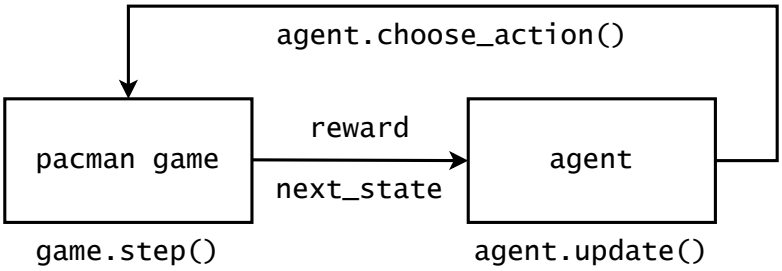

图 4　Q-Learning 流程图

TD 更新在 `update()` 方法中实现。给定转移 `(state, action, next_state, reward)`，算法首先计算 TD 目标和 TD 误差：

```
td_target = reward + discount * get_state_value(next_state)
td_delta = td_target - get_q_value(state, action)
```

其中 `get_state_value(s)` 返回 $\max_{a'} Q(s, a')$。随后更新 Q 值：

```
Q_values[(state, action)] += alpha * td_delta
```

这一实现对应算法 2 的第 10 行，直接应用式 (6)。

环境在每次调用 `step()` 后会触发 `observe_transition()` 方法，该方法计算奖励并调用 `update()` 完成 Q 值更新。这种设计使得算法能够在线学习，无需维护 episode 缓冲区。

3.4 Approximate Q-Learning 实现

Approximate Q-Learning 通过继承 Q-Learning 并重写关键方法实现。核心改变在于用权重向量替代 Q 表。

权重向量同样使用 Python 字典实现：`weights: dict[feature_name] -> float`。字典的键为特征名称（字符串），值为对应的权重。这一设计允许特征维度动态扩展，且未出现的特征默认权重为 0。

特征提取通过 `FeatureExtractor` 类实现。该类的 `get_features(state,`

action) 方法返回特征字典：dict[feature_name] -> value。特征提取器根据表 1 的设计，从状态对象中查询相关信息并计算特征值。例如，closest-food 特征通过广度优先搜索计算最短路径距离，然后归一化。

Q 值计算通过特征与权重的内积实现：

```
def get_q_value(state, action):
    features = feat_extractor.get_features(state, action)
    return sum(weights[f] * features[f] for f in features)
```

这对应式 (7) 的线性组合 $\mathbf{w}^\top \mathbf{f}(s, a)$。

权重更新在 update() 方法中实现。算法首先计算 TD 误差，然后对每个非零特征更新对应权重：

```
td_target = reward + discount * get_state_value(next_state)
td_delta = td_target - get_q_value(state, action)


for f in features:
    weights[f] += alpha * td_delta * features[f]
```

这一实现对应算法 3 的第 10-11 行和式 (9)。

相比表格式方法，函数近似方法的内存占用显著降低。表格式方法可能存储数万个状态-动作对，而函数近似仅需存储约 10 个特征权重。这一优势在大规模地图上尤为明显。

所有算法均实现了模型保存与加载功能，使用 Python 的 pickle 模块序列化 Q 表或权重向量。训练完成后，可将模型保存为文件，测试时直接加载，无需重新训练。

4 实验与分析

本章介绍实验设置及结果分析。其中，由于 Monte Carlo Learning 和 Q-Learning 受限于状态空间爆炸问题，仅在小规模地图上进行测试；主要对 Approximate Q-Learning 算法进行了详细的对比实验和消融实验，以验证其有效性，并分析特征设计对性能的影响。

4.1 实验设置

4.1.1 实验环境

实验使用多种规模的地图进行测试。填表式方法（MC learning 和 Q-Learning）由于状态空间爆炸问题，仅在小规模地图 `smallGrid`（7×7，约 3 个食物）上进行训练和测试。该地图包含 1 个鬼，没有胶囊。

Approximate Q-Learning 则在更大规模的地图上进行实验，包括：`smallClassic`（20×7，约 28 个食物），`mediumClassic`（20×11，约 48 个食物），`originalClassic`（28×27，约 240 个食物）。这些地图均包含 2 个随机移动的鬼和若干胶囊（`small` 和 `medium` 中有 2 个，`original` 中有 4 个）。鬼的移动策略为随机选择合法动作，攻击概率和逃跑概率均设为 0.2，引入一定的随机性。图 5 展示了三种 Classic 地图的结构。

图 5 实验使用的三种 Classic 地图：`smallClassic`（左）、`mediumClassic`（中）、`originalClassic`（右）

4.1.2 参数配置

三种算法采用相同的超参数配置。学习率 $\alpha = 0.1$，探索率 $\epsilon = 0.05$，折扣因子 $\gamma = 0.8$。MC learning 和 Q-Learning 在 `smallGrid` 上训练 500 episode。Approximate

Q-Learning 在三种 Classic 地图上分别训练 1000 episode。每个 episode 的最大步数限制为 1000 步，超过则强制终止。

4.1.3 评价指标

实验采用以下指标评估算法性能。平均分数（Average Score）衡量智能体获得的平均总分，反映整体表现；胜率（Win Rate）表示成功吃掉所有食物的 episode 比例，反映任务完成能力；最高分数（Highest Score）记录测试中获得的最高分数，反映算法的性能上限。所有指标均在训练完成后的 100 个测试 episode 上统计，测试时设置 $\epsilon = 0$ 、$\alpha = 0$。

4.2 对比实验结果与分析

4.2.1 表格式方法在小规模地图上的表现

表 2 展示了 MC learning 和 Q-Learning 在 `smallGrid` 地图上的测试结果。

表 2 表格式方法在 `smallGrid` 地图上的性能（训练 500 episode）

算法	平均分数	胜率 (%)	最高分数
MC Learning	-7.8	50.0	499.0
Q-Learning	436.0	94.0	500.0

Q-Learning 在 `smallGrid` 上取得了 94% 的胜率和 436.0 的平均分数，表明在状态空间较小的情况下，表格式方法能够有效学习最优策略。MC Learning 的胜率为 50%，平均分数为 -7.8，性能显著低于 Q-Learning。这可能是因为 Q-Learning 采用单步 TD 更新，能够快速传播价值信息，而 MC learning 需要等待完整 episode 才能更新，在长 episode 场景下学习效率较低，且对初期探索的依赖性更强。

然而，表格式方法的局限性在于无法扩展到更大规模的地图。当尝试在 `smallClassic` 等地图上训练时，由于状态空间过大（约 10^{17}），智能体在有限的训练时间内无法充分访问所有状态，导致学习失败。这一问题促使引入函数近似方法。

4.2.2 Approximate Q-Learning 在不同规模地图上的表现

表 3 展示了 Approximate Q-Learning 在三种不同规模地图上的测试结果。

从实验结果可以观察到，在 `smallClassic` 和 `mediumClassic` 地图上，

表 3　Approximate Q-Learning 在不同地图上的性能（训练 1000 episode）

地图	平均分数	胜率 (%)	最高分数
`smallClassic`	1332.1	89.0	1778.0
`mediumClassic`	1602.1	84.0	2148.0
`originalClassic`	2470.5	62.0	3834.0

Approximate Q-Learning 分别取得了 89% 和 84% 的胜率，表现稳定。通过特征提取和线性函数近似，智能体能够有效泛化到未见过的状态，克服了表格式方法在大状态空间下的局限性。

随着地图规模继续增大，算法在 `originalClassic` 地图上的胜率降低至 62%，相比小地图有所下降但仍保持较好的性能。该地图的状态空间和食物数量远超训练环境（约 240 个食物），且地图结构更加复杂、鬼的数量更多。这一结果表明函数近似方法具有较强的泛化能力。

4.3　消融实验结果与分析

4.3.1　函数近似方法的组件消融

为验证 Approximate Q-Learning 中各组件的作用，进行消融实验。所有实验均在 `smallClassic` 地图上训练 1000 episode。

表 4　Approximate Q-Learning 消融实验结果

方法配置	平均分数	胜率 (%)	最高分数
完整方法	1332.1	89.0	1778.0
移除线性逼近	130.2	24.0	1310.0
移除特征提取		无法训练	

实验结果表明，线性函数近似对性能至关重要。移除线性逼近后，仍使用特征向量表示状态，但采用表格式存储每个特征组合的 Q 值，性能大幅下降：平均分数从 1332.1 降至 130.2，胜率从 89% 降至 24%。这是因为表格式方法无法泛化到未见过的特征组合。相比之下，线性函数近似通过 $Q(s,a) = \mathbf{w} \cdot \mathbf{f}(s,a)$ 实现特征空间的插值，即使特征组合未在训练中出现，也能根据相似特征的权重给出合理估计。

移除特征提取后，算法相当于在原始状态空间上应用线性逼近。由于 `smallClassic` 的状态空间过大（约 10^{17}），智能体在有限训练时间内无法充分探索，导致学习失败。这一结果也验证了特征提取在降维中的必要性。

4.3.2 特征设计的影响

在 Approximate Q-Learning 基础上，进一步实验不同特征配置对性能的影响，重点分析标志位特征的作用。实验均使用 `smallClassic` 地图，训练 1000 episode。

表 5 特征配置对比实验

特征配置	平均分数	胜率 (%)	最高分数
完整特征集	1332.1	89.0	1778.0
移除所有标志位特征	759.9	82.0	990.0
仅移除食物标志位	645.8	70.0	1374.0

实验结果表明，标志位特征对性能影响显著。移除所有标志位特征（`eats-food`, `eats-scared-ghost`, `eats-capsule`）后，平均分数从 1332.1 下降至 759.9，胜率从 89% 下降至 82%。进一步分析发现，仅移除食物标志位的影响更大，胜率降至 70%，平均分数降至 645.8。

通过可视化观察，移除标志位特征后，智能体即使路过胶囊也不会主动吃掉。这可能是因为胶囊本身不直接给分，仅通过距离特征无法学习到吃胶囊的长期价值；而标志位特征提供了即时反馈，帮助智能体识别关键动作（如吃食物、吃胶囊、吃恐慌的鬼）的价值，从而更快地学习有效策略。

4.4 训练过程观察

4.4.1 首次获胜现象分析

实验中观察到，Approximate Q-Learning 在训练初期（通常前 2-3 个 episode）即可首次取得胜利。然而通过可视化分析发现，此时智能体的策略尚不成熟。

具体表现为：智能体不会主动寻找远处的食物，当视野范围内无食物时倾向于停留或原地徘徊；仅在遇到鬼追赶时被动移动，若在移动过程中碰到食物才会继续得分。这种早期获胜依赖于环境的随机性，智能体尚未学到系统性的搜索

策略。

随着训练的进行，智能体逐渐学会主动规划路径、高效收集食物，胜率和平均分数稳步提升，最终收敛到稳定的性能水平。这一现象反映了强化学习中探索与利用的平衡过程：初期随机探索可能偶然获得高回报，但需要通过持续学习才能形成稳定的最优策略。

4.4.2 特征权重的变化

训练过程中，特征权重的变化反映了智能体学习策略的过程。图 6 展示了主要特征权重在 `smallClassic` 地图 1000 轮训练中的变化趋势，按权重变化幅度分为三组。

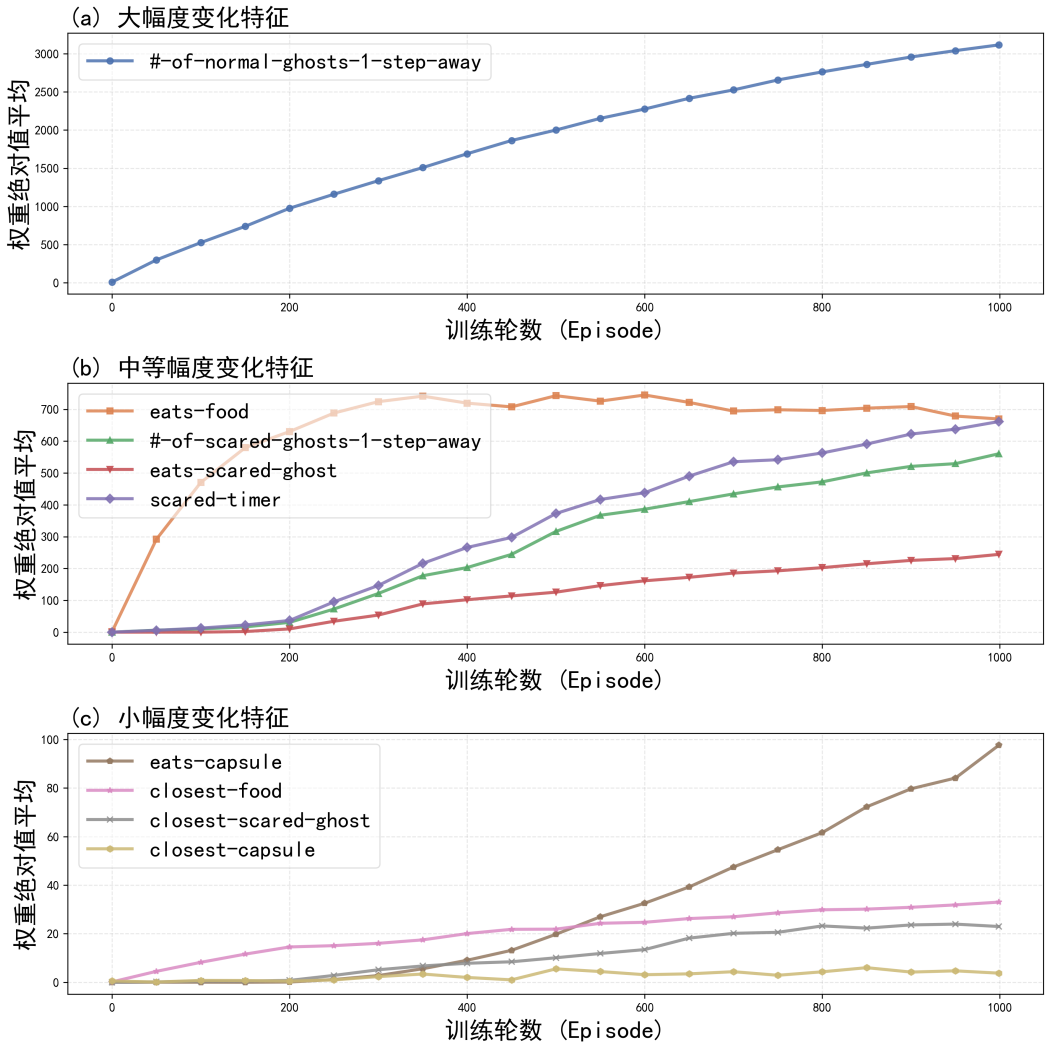

图 6　主要特征权重随训练轮数的变化

从权重变化曲线可以观察到以下现象。

图 (a) 展示了大幅度变化的特征。#-of-normal-ghosts-1-step-away 权重绝对值持续增长至超过 2500，且在整个训练过程中未见饱和趋势，表明回避危险鬼是学习的核心策略。

图 (b) 展示了中等幅度变化的特征。eats-food 权重在训练初期快速增长，约 500 轮后稳定在 730 左右，说明智能体优先学会获取即时奖励。#-of-scared-ghosts-1-step-away 和 scared-timer 权重持续增长至约 500 和 550，反映了智能体逐渐学会利用鬼受惊状态的战术优势。eats-scared-ghost 权重在训练约 150 轮后才开始明显增长，最终达到约 210，说明智能体在掌握基本生存策略后才学会这一进阶技巧。

图 (c) 展示了小幅度变化的特征。距离类特征（closest-food、closest-scared-ghost、closest-capsule）和 eats-capsule 的权重整体呈增长趋势，但变化幅度远小于其他特征。这些特征的权重变化模式与预期存在差异，可能与特征设计或训练动态有关，具体原因尚不明确。

总体而言，权重变化呈现分阶段学习的特点：训练初期优先学会即时收益和危险回避，中期开始利用鬼受惊状态，后期逐渐优化长期规划。

5 总结

从课堂、教材和网络资料中可以理解强化学习算法的数学原理和理论流程，但真正去实现代码、调试训练过程时仍会遇到许多问题。而在修改代码的过程中，对于状态表示、动作选择、奖励设计等核心概念也会有新的认识。

实验中在环境搭建和特征设计部分花费的时间最长，主要是最初并不理解强化学习环境的接口规范，也不清楚状态空间和动作空间应该如何定义。通过查找 Gymnasium 文档了解标准接口的要求；通过逐步输出观察状态从原始游戏数据到特征向量的转换过程，理解智能体如何感知环境；对照各算法的更新公式，逐渐摸清 Q 值表、特征权重等数据结构的作用，之后才顺利完成算法实现。

在实现表格式方法（MC Learning、Q-Learning）时发现，即使是小规模的 7×7 地图，状态空间已经非常庞大，训练时很难遍历所有状态。这让我真正体会到”维度灾难”不仅是理论上的概念，在实际应用中会严重制约算法的可行性。而在引入函数近似、并通过手工设计特征将高维状态压缩到低维空间后，智能体仅用 1000 个 episode 就学会了有效的策略，并且有很强的泛化能力。这让我深刻认识到算法和特征设计的重要性。

在特征设计过程中收获颇丰，通过分析权重演化发现智能体会优先学习危险回避，然后才学会利用恐慌鬼获取高分，这种分阶段学习的现象很有意思。但遗憾的是没有足够的时间去尝试更多的算法变体，在调试代码细节上花了比较多的时间，接触新方法较少。此外，当前特征主要基于简单的距离计算，对于路径规划、区域控制等更高层次的策略缺乏建模，导致在复杂地图上的表现还不够理想。

未来还可以尝试深度强化学习方法（如 DQN），用神经网络自动学习特征，避免手工设计的局限性；以及引入更复杂的训练技巧，如经验回放、目标网络等，提高样本利用效率；

总体而言，本次实验让我对强化学习有了更深入的理解，不仅停留在理论层面，而是通过实践体会到了算法设计、环境建模、特征工程的重要性和挑战性。