基于强化学习的无人机飞跃冰湖实验

傅信燊，秦华谦，向航，张祖烨（拼音顺序）*，古博老师†

中山大学 智能工程学院，深圳 518000

【摘　要】 本次机器人综合实验的主题是无人机飞越冰湖。为了解决传统无人机避障算法中需要构建离线三维地图的问题，实验采用了基于马尔可夫决策过程的 Qlearning、Sarsa 和 DQN 算法。在 Deterministic 模式和 Stochastic 模式下的 4x4 及 8x8 地图上进行了相应的强化学习（RL）模型训练。通过性能比较，成功地在实机演示中控制无人机安全地飞越冰湖。

【关键词】 强化学习，冰湖问题，无人机

UAV flying over a frozen lake experiment based on reinforcement learning

Fu Xinshen, Qin Huaqian, Xiang Hang, Zhang Zuyi， Prof. Gu Bo

School of Intelligent Engineering, Sun Yat-sen University, Shenzhen 518000, China

Abstract: The theme of this comprehensive robot experiment was the use of drones to fly over an ice lake. To address the issue of traditional drone obstacle avoidance algorithms requiring the construction of offline 3D maps, the experiment employed Qlearning, Sarsa, and DQN algorithms based on Markov Decision Processes. These algorithms were trained on corresponding Reinforcement Learning (RL) models on 4x4 and 8x8 maps under both Deterministic and Stochastic modes. Through performance comparisons, the experiment successfully demonstrated the control of drones flying over an ice lake in real machine demonstrations.

Key Words: Reinforcement learning, glacial lake problems, UAV

1　概 述

小型多旋翼无人机因其小巧灵活的特性，适宜在复杂环境中执行机动任务，因此在军用和民用领域得到了广泛应用。随着科技的进步，针对无人机的导航、指导和控制技术也获得了显著提升。然而，实时进行自主导航尤其在未知和结构不确定的三维环境中，尤其是在变化多端的光照条件下，仍然是一个极具挑战性的问题。这些环境中实现自主避障仍是一个迫切需要解决的问题。

图 1 小型四旋翼无人机

传统的无人机避障算法依赖于构建离线三维地图，并在此基础上使用路径搜索算法来规划最优路径，这些路径以障碍物为约束。尽管一些避障

实验时间: 2023-11-27

报告时间: 2023-11-30

† 指导教师：古博

*学号：21312463 21312683 21312505 20354171

*E-mail：qinhq5@mail2.sysu.edu.cn

算法避免了复杂的地图构建工作，它们仍需要手动调整大量参数，且在避障过程中无法利用经验进行自我迭代。随着机器学习的发展，研究人员开始将有监督学习应用于无人机避障，把避障任务视作一个基于监督学习的分类问题，但这需要对每个样本进行耗时的标注工作。

在无人机避障过程中，如何有效利用三维空间信息，避免繁琐的环境构建工作，简化算法模型，以及提升无人机的自主避障能力和效率，从而更安全、高效地到达指定目的地，仍是当前亟需解决的关键问题。

随着机器学习技术的进步，强化学习已经被科研人员应用于无人机控制领域。强化学习通过与外部环境的交互来优化行为，相比于传统机器学习方法，它具有以下优势：

1. 在训练前，强化学习不需要对数据进行标注处理，这使得它能更有效地处理环境中的特殊情况；

2. 强化学习可以将整个系统视为一个统一整体，从而实现端到端的输入输出。这使得系统的某些模块具有更强的鲁棒性；

3. 相比其他机器学习方法，强化学习更容易学习一系列的行为。

使用强化学习算法的优势在于它不依赖于传统非机器学习方法所需的离线地图，也不依赖于监督学习中所需的人工标注数据集。通过深度学习模型，强化学习学习输入数据与输出行为之间的映射关系，从而使智能体能够处理高维连续空间中的决策问题，并避免了复杂的离线地图构建工作。

基于这些优势，我们的研究小组通过查阅资料，对多种强化学习算法在冰湖环境下进行了仿真实验和性能对比。最终，我们选择了表现最佳的 Sarsa 算法，并成功应用于实际无人机飞越冰湖的演示中。

图 2 冰湖实验环境

2 算法介绍

2.1 强化学习（RL）

强化学习（RL）是一种机器学习方法，它使得智能体（agent）通过与环境的交互来学习如何达到一个或多个目标。这种学习方式与人类或动物学习经验的过程有相似之处，即通过试错的方式逐渐适应环境并实现目标。

强化学习过程中的一些关键概念如下：

- 智能体（Agent）：在 RL 中，智能体是进行决策的实体，它选择执行哪些行为（actions）。
- 环境（Environment）：智能体所处并与之交互的外部世界或系统。
- 状态（State）：环境在任何特定时间点的状态。智能体的决策通常基于环境的当前状态。
- 行为（Action）：智能体在特定状态下可以执行的操作。
- 回报（Reward）：智能体执行特定行为后，环境提供的即时反馈。这是评价智能体行为好坏的标准。
- 策略（Policy）：策略是从状态到行为的映射。它定义了智能体在特定状态下应该采取的行为。

agent 通过与环境交互来进行学习。这是一个以目标为导向的学习过程，agent 从其行为的结果中学习。agent 可以根据最好的结果探索不同行

为，也可以利用获得了良好回报的先前行为。如果 agent 探索不同的行为（action），则很有可能 agent 将获得较差的回报，因为所有的行为并不都是最佳的。如果 agent 仅利用已知的最佳行为，那么错过最佳行为（action）的可能性也很大。因此，需要选择合适的算法在这两者间取得平衡。一个典型 RL 算法的步骤如下:

1. 首先，agent 通过执行行为与环境进行交互;
2. agent 执行行为并从一种状态移动到另一种状态;
3. 然后 agent 将根据其执行的行为获得回报;
4. 根据回报，agent 将知道行为是好的还是坏的;
5. 如果 action 是好的，也就是说，如果 agent 得到了积极的 reward，那么 agent 将更喜欢执行该 action，否则 agent 将尝试执行导致积极 reward 的其他 action。

强化学习作为一种机器学习方法，其独特性在于它强调在不确定和复杂的环境中通过交互学习，使得它在实时决策和自适应控制方面尤为有效。随着算法和计算能力的不断发展，强化学习未来在更多领域的应用前景看好。

2.2 Q-Learning

在强化学习中，智能体（agent）通过与环境的交互来学习如何最大化累积奖励。Q-Learning 是强化学习中的一种重要算法，其名称来源于它使用的一个核心组件——Q 表。[1]

Q-Table		Actions				
		Action 1	Action 2	...	Action N-1	Action N
States	State 1	0.236512	0.165112	...	0.542213	0.45458
	State 2	0.681424	0.445845	...	0.212424	0.554712
	...	...	...	...	...	...
	State M-1	0.788554	0.155514	...	0.554588	0.224245
	State M	0.788954	0.554845	...	0.225255	0.742657

图 3 Q 表示例

Q 表是一种数据结构，用于存储和更新每个状态-动作对的值（Q 值），Q 即为 $Q(s,a)$，，就是在某一时刻的 s 状态下 $s(s \in S)$ 采取动作 $a(a \in A)$ 动作能够获得收益的期望，环境会根据 agent 的动作反馈相应的回报 reward r，所以算法的主要思想是 Q 表随着状态、动作而更新，当 Q 表更新不再发生改变时，就可以根据环境选择对应最大的值所对应的动作，从而采取动作，进行与环境交互。

Q 表的更新是通过一种称为时间差分学习的方法进行的。在每一步中，智能体根据当前的 Q 表选择一个行为，执行该行为后，它接收到一个奖励和新的状态信息。然后，使用这个新信息来更新 Q 表中当前状态和行为的 Q 值。

Q-learning 算法的更新准则如下式所示:

$$Q[S,A] = (1-\alpha)*Q[S,A]+\alpha*(R+\gamma*\max Q[S_n ext;])$$

其中，α 是学习率，决定了新信息覆盖旧信息的速度；R 是即时奖励；γ 是折扣因子，用于平衡即时奖励和未来奖励；$S_n ext$ 是新状态；

构建 Q 表算法步骤：

Step 1: 给定参数 γ 和 reward 矩阵 R

Step 2: 令 $Q := 0$

Step 3: For each episode:

3.1 随机选择一个初始的状态 s

3.2 若未达到目标状态，则执行以下几步：

1. 在当前状态 s 的所有可能行为中选取一个行为 α
2. 利用选定的行为 α，得到下一个状态 $\tilde{s}$
3. 按照算法更新公式计算 $Q(s,a)$
4. 令 $s := \tilde{s}$

在 Q-Learning 中，智能体的策略通常是一种贪婪策略，即在大多数情况下选择当前 Q 值最高的行为。然而，为了鼓励探索，通常会加入一定比例的随机行为选择，这称为 ϵ-贪婪策略。

Agent 利用上述算法从经验中进行学习。每一个 episode 相当于一个 training session。在一个 training session 中，agent 探索外界环境，并接收外界环境的 reward，直到达到目标状态。训练的目的是要强化 agent 的“大脑”（用 Q 表示）。训练得越多，则 Q 被优化得更好. 当矩阵 Q 被训练强化后，agent 便很容易找到达到目标状态的最快路径了。

利用训练好的矩阵 Q，我们可以很容易地找出一条从任意状态 s_0 出发达到目标状态的行为路径，具体步骤如下：

1. 令当前状态 $s := s_0$

2. 确定 a，它满足 $Q(s,a) = \max_{\tilde{a}}\{Q(s,\tilde{a})\}$.

3. 令当前状态 $s := \tilde{s}$（$\tilde{s}$ 表示 a 对应的下一个状态）

4. 重复执行步 2 和步 3 直到 s 成为目标状态

当 Q 表中的值不再发生显著变化时，我们认为算法已经收敛。此时，Q 表提供了在每个状态下采取每个可能行为的最优策略。

通过使用 Q-Learning，智能体能够学习在各种不同的状态下采取何种行为以最大化总奖励。这种学习方法在处理复杂、未知或变化的环境中显示出了巨大的潜力，尤其是在那些难以用传统算法编程解决的问题上。

2.3 Srasa

Sarsa[2]这个名字来源于五个关键因子：S（State 当前状态）、A（Action 当前行动）、R（Reward 即时奖励）、S（State 下一时刻的状态）和 A（Action 下一时刻的行动）。

Sarsa 算法的的决策部分与 Q-learning 相同，都是通过 Q 表的形式进行决策，在 Q 表中挑选值较大的动作值施加在环境中来换取奖惩，也就是根据计算出来的 Q 值来作为选取动作的依据，两者不同的是行为更新准则是有差异的。在 Q-Learning 中，智能体会选择它认为可能获得最大 Q 值的下一个行为，即使这个行为并未实际执行。而在 Sarsa 算法中，智能体更新当前行为的 Q 值时，使用的是实际执行的下一个行为的 Q 值。（Sarsa 不会去选取他估计出来的最大 Q 估计值，而是直接选取估计出来的 Q 值）。

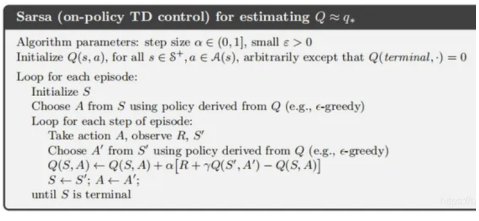

图 4 Sarsa

Sarsa 算法的更新准则如下式所示：

$$Q[S,A] = (1-\alpha)*Q[S,A] + \alpha*(R + \gamma*Q[S_{next}, A_{next}])$$

其中，S 和 A 分别代表当前状态和行为，R 是获得的即时奖励，γ 是折扣因子，S_{next} 和 A_{next} 分别是下一个状态和实际采取的下一个行为。

具体的改变就是在于:

$\max Q[S_next;]$ 变成了 $Q[S_{next}, A_{next}]$

Sarsa 算法的这种更新方式使得其更加注重实际执行的路径，而不是仅仅追求理论上可能的最大回报。这种方式使得 Sarsa 在某些情况下比 Q-Learning 更加稳健，因为它考虑到了实际的状态转移和行为选择。

在实际应用中，这意味着 Sarsa 算法倾向于选择一条相对安全的路径，尽可能避免陷阱和不确定性。相比之下，Q-Learning 更倾向于追求最快的路径，即使这条路径可能涉及更多的风险。这种差异在某种程度上提高了 Q-Learning 的泛化能力，但也可能使其陷入局部最优解，而忽视了更安全或可靠的解决方案。

总的来说，Sarsa 算法通过其独特的更新准则，提供了一种更加注重实际执行路径的学习策略，这在复杂和不确定的环境中可能更为有效。

2.4 DQN

在 Q-learning 算法中，我们以矩阵的方式建立了一张存储每个状态下所有动作值的表格。表格中的每一个动作价值表示在状态下选择动作，然后继续遵循某一策略预期能够得到的期望回报。然而，这种用表格存储动作价值的做法只在环境的状态和动作都是离散的，并且空间都比较小的情况下适用。当状态或者动作数量非常大的时候，这种做法就不适用了。极端情况下，当状态或者动作连续的时候，就有无限个状态动作对，我们更加无法使用这种表格形式来记录各个状态动作对的 Q 值。

对于这种情况，我们需要用函数拟合的方法来估计值，即将这个复杂的值表格视作数据，使用一个参数化的函数来拟合这些数据。很显然，这种函数拟合的方法存在一定的精度损失，因此被称为近似方法。DQN 算法便可以用来解决连续状态下离散动作的问题。

由于神经网络具有强大的表达能力，因此我们可以用一个神经网络来表示函数 Q。[3] 在动作是离散（有限）的情况下，除了可以采取动作连续情况下的做法，我们还可以只将状态输入到神经网

络中，使其同时输出每一个动作 Q 的值。

假设神经网络用来拟合函数的参数是，即每一个状态下所有可能动作 a 的 Q 值我们都能表示为 $Q_\omega(s,a)$。我们将用于拟合函数函数的神经网络称为 Q 网络.

那么 Q 网络的损失函数是什么呢？我们先来回顾一下 Q-learning 的更新规则

$$Q[S,A]=(1-\alpha)*Q[S,A]+\alpha*(R+\gamma*\max Q[S_next;])$$

上述公式用时序差分（temporal difference，TD）学习目标 $R+\gamma*\max Q[S_next;]$ 来增量式更新 $Q[S,A]$，也就是说要使和 TD 目标 $R+\gamma*\max Q[S_next;]$ 靠近。于是，对于一组数据，我们可以很自然地将 Q 网络的损失函数构造为均方误差的形式：

$$\omega^*=\arg\min_\omega \frac{1}{2N}\sum_{i=1}^{N}\left[Q_\omega(s_i,a_i)-\left(r_i+\gamma\max_{a'}Q_\omega(s_i',a')\right)\right]^2$$

至此，我们就可以将 Q-learning 扩展到神经网络形式——深度 Q 网络（deep Q network，DQN）算法。由于 DQN 是离线策略算法，因此我们在收集数据的时候可以使用一个 ϵ-贪婪策略来平衡探索与利用，将收集到的数据存储起来，在后续的训练中使用。DQN 中还有两个非常重要的模块——经验回放和目标网络，它们能够帮助 DQN 取得稳定、出色的性能。

经验回放（评估网络）：

在一般的有监督学习中，假设训练数据是独立同分布的，我们每次训练神经网络的时候从训练数据中随机采样一个或若干个数据来进行梯度下降，随着学习的不断进行，每一个训练数据会被使用多次。在原来的 Q-learning 算法中，每一个数据只会用来更新一次值。为了更好地将 Q-learning 和深度神经网络结合，DQN 算法采用了经验回放（experience replay）方法，具体做法为维护一个回放缓冲区，将每次从环境中采样得到的四元组数据（状态、动作、奖励、下一状态）存储到回放缓冲区中，训练 Q 网络的时候再从回放缓冲区中随机采样若干数据来进行训练。这么做可以起到以下两个作用。

（1）使样本满足独立假设。在 MDP 中交互采样得到的数据本身不满足独立假设，因为这一时刻的状态和上一时刻的状态有关。非独立同分布的数据对训练神经网络有很大的影响，会使神经网络拟合到最近训练的数据上。采用经验回放可以打破样本之间的相关性，让其满足独立假设。

（2）提高样本效率。每一个样本可以被使用多次，十分适合深度神经网络的梯度学习。

目标网络：

DQN 算法最终更新的目标是让 $Q_\omega(s,a)$ 逼近 $R+\gamma*\max Q[S_next;]$，由于 TD 误差目标本身就包含神经网络的输出，因此在更新网络参数的同时目标也在不断地改变，这非常容易造成神经网络训练的不稳定性。为了解决这一问题，DQN 便使用了目标网络（target network）的思想：既然训练过程中 Q 网络的不断更新会导致目标不断发生改变，不如暂时先将 TD 目标中的 Q 网络固定住。为了实现这一思想，我们需要利用两套 Q 网络。

（1）原来的训练网络 $Q_\omega(s,a)$，用于计算原来的损失函数中 $\frac{1}{2}[Q_\omega(s,a)-(r+\gamma\max_{a'}Q_{\omega^-}(s',a'))]^2$ 的项，并且使用正常梯度下降方法来进行更新。

（2）目标网络 $Q_{\omega^-}(s,a)$，用于计算原先损失函数 $\frac{1}{2}[Q_\omega(s,a)-(r+\gamma\max_{a'}Q_{\omega^-}(s',a'))]^2$ 中的 $(r+\gamma\max_{a'}Q_{\omega^-}(s',a'))$ 项，其中 ω^- 表示目标网络中的参数。如果两套网络的参数随时保持一致，则仍为原先不够稳定的算法。为了让更新目标更稳定，目标网络并不会每一步都更新。具体而言，目标网络使用训练网络 $Q_\omega(s,a)$ 的一套较旧的参数，训练网络在训练中的每一步都会更新，而目标网络的参数每隔 C 步才会与训练网络同步一次，即 $\omega^-\leftarrow\omega$。这样做使得目标网络相对于训练网络更加稳定。

具体来讲，流程如下：

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^-=\theta$
For episode = 1, M **do**
  Initialize sequence $s_1=\{x_1\}$ and preprocessed sequence $\phi_1=\phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t=\text{argmax}_a Q(\phi(s_t),a;\theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1}=s_t,a_t,x_{t+1}$ and preprocess $\phi_{t+1}=\phi(s_{t+1})$
    Store transition $(\phi_t,a_t,r_t,\phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j,a_j,r_j,\phi_{j+1})$ from D
    Set $y_j=\begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j+\gamma\max_{a'}\hat{Q}(\phi_{j+1},a';\theta^-) & \text{otherwise}\end{cases}$
    Perform a gradient descent step on $\left(y_j-Q(\phi_j,a_j;\theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q}=Q$
  End For
End For

图 5 DQN 算法流程

2.5 Deterministic vs. stochastic

当环境的下一个状态是由当前的状态以及 agent 要执行的 action 决定的，我们说它是 Deterministic，否则，说它是 stochastic。在一个“完全可见的”，且“Deterministic”环境中，agent 不需要担心不确定的东西。比如在游戏中，agent 是 Deterministic，即使它不能预测其他玩家的行动。而在现实中，形势太复杂，我们必须得看作是 stochastic，因为我们不可能知道所有不可见的因素。如果一个环境不是完全可见的，或者不是确定性的，我们说它是 uncertain。“stochastic”一般来说意味着后果的不确定性在概率上是可以量化的。nondeterministic 的环境是指 action 以它们可能的后果为特点，但是没有概率属性。nondeterministic 的环境通常用于描述衡量 performace 时 agent 达到 action 的所有可能的后果。

具体来讲，就是在随机环境中，状态转移和奖励不仅取决于当前状态和行为，还受到随机因素的影响。这意味着对于相同的状态和行为，每次的结果（下一个状态和奖励）可能不同。而在确定性环境中，给定当前状态和行为，下一个状态和奖励是完全确定的。

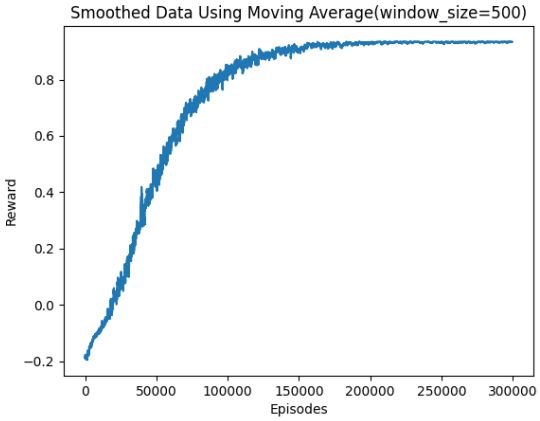

图 6 $Q-learning$

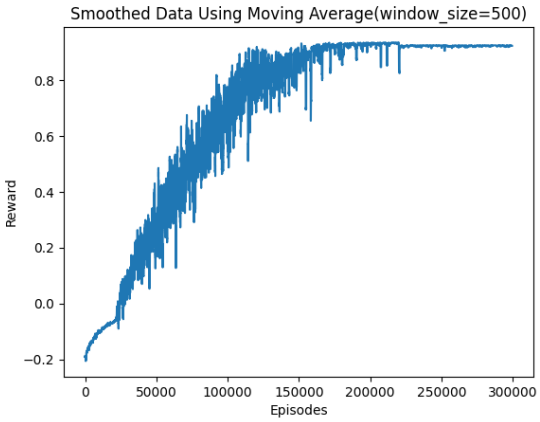

图 7 Sarsa

3 实验结果与分析

3.1 Deterministic 模式

在 Deterministic 模式下的实验中，我们分别对 $Q-learning$ 和 Sarsa 两种强化学习算法进行了详细的比较和分析。这两种算法都在处理相同的任务，并在相同条件下运行了 300,000 个 Episodes。通过绘制它们的收敛曲线，我们能够深入了解这两种算法的学习效率和行为特性。

通常每个 Episode 的奖励是检查收敛的最直接指标。如果奖励随时间变得稳定（不再有显著波动），这意味着算法正在收敛。因此使用滑动窗口平均（$window_size = 500$）的方法，在 episode 为 300,000 的情况下，记录每个 episode 的全局 rewards，将其视作模型的训练效果。作图如下：

收敛曲线表明，Q-learning 和 Sarsa 算法都在大约 150,000 到 200,000 个 Episodes 之间达到了收敛状态。这一现象表明，尽管两种算法在学习策略上有所不同，它们在这个特定的任务中都能在相似的时间范围内学习到有效的策略。然而，值得注意的是，多次实验并对比以后发现，Q-learning 算法展现出了相对更快的收敛速度。

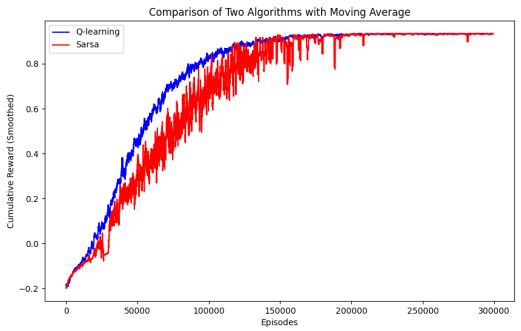

图 8 compare1

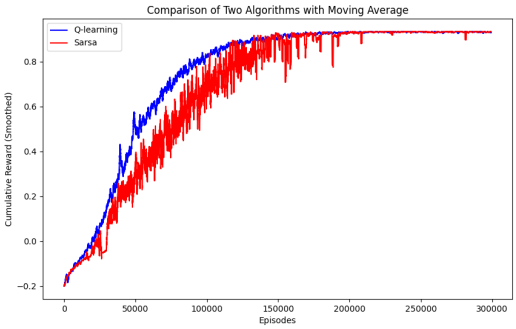

图 9 compare2

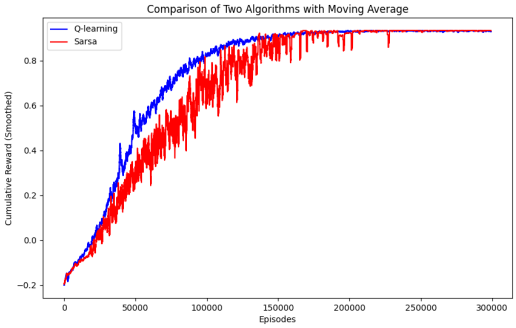

图 10 compare3

除了收敛速度外，我们还对两种算法的运行效率进行了对比。运行时间的记录显示，在完成 300,000 个 Episodes 的任务中，Sarsa 算法需要 2 分钟 5 秒，而 Q-learning 算法仅需要 1 分钟 40 秒。这个时间差异虽然不是特别显著，但在大量 Episodes 累积下来的情况下，这种效率的提升仍然是值得关注的。Q-learning 算法更短的运行时间表明它在处理这类任务时可能更为高效。这可能与其在更新价值估计时更为直接和积极的策略有关。

总体来看，在 Deterministic 模式下，Q-learning 算法在收敛速度和运行效率上均表现出一定的优势。这些发现对于选择和优化强化学习算法在特定任务和环境下的应用提供了重要的参考依据。

3.2 Stochastic 模式

在 Stochastic（随机）模式下，对 Q-learning 和 Sarsa 两种强化学习算法进行了类似的实验和比较分析。与 Deterministic（确定性）模式相比，Stochastic 模式下的环境为算法带来了新的挑战。在这种模式下，我们也绘制了两种算法的收敛曲线进行观察。

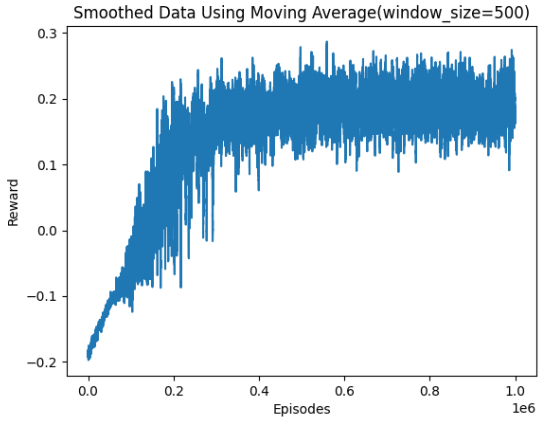

图 11 Q-learning with Stochastic

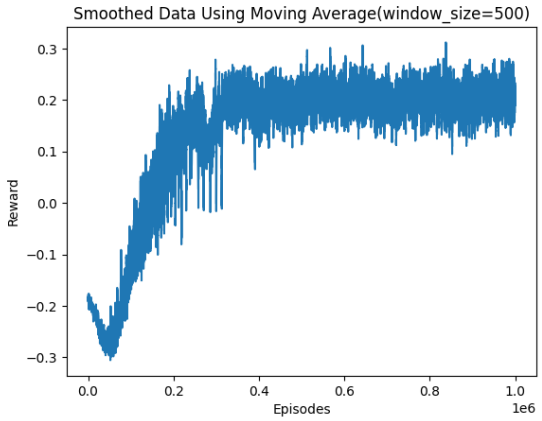

图 12 Sarsa with Stochastic

不同于 Deterministic 模式，我们发现在 Stochastic 模式下，两种算法在收敛过程中表现出更大的振荡，尤其是 Q-learning 算法。这种振荡表明了较差的稳定性，尤其是在算法的后期阶段。在 Stochastic 模式中，代理执行动作后，环境的状态转移涉及到一定的随机性。这意味着状态的转换不仅取决于代理的行动，还受到环境因素的影响，比如风力或冰面打滑等。这与 Deterministic 模式形成鲜明对比，在后者中，环境状态的转换完全由代理的行动决定。

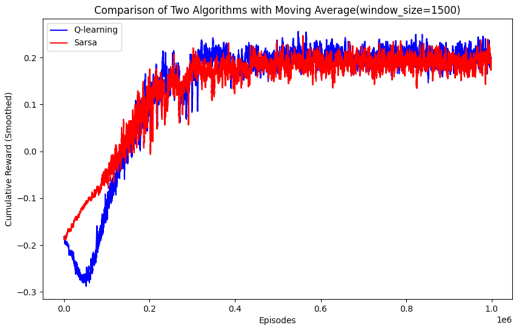

图 13 Stochastic Mode Compare

由于 Stochastic 模式下状态转移的随机性，下一个状态 s_{t+1} 不仅仅与当前动作和当前状态 s_t 有关，这增加了代理的学习难度。这种随机性使得算法难以稳定地学习一个稳定有效的策略，因此，收敛变得更为困难。特别是对于 Q-learning 算法，这种挑战似乎更加显著，因为其学习过程依赖于对未来奖励的估计，而在随机环境中这种估计更加困难和不稳定。

Sarsa 算法，作为一种在策略内学习的方法，在面对这种随机性时，可能表现出相对更好的适应性，因为其学习过程更加注重当前策略下的期望奖励，而不是像 Q-learning 那样依赖于最大化未来奖励的估计。

总体来说，在 Stochastic 模式下，环境的不确定性和随机性给强化学习算法的稳定性和收敛带来了额外的挑战，这在 Q-learning 算法中表现得尤为明显。这些挑战需要通过改进算法或采用不同的策略来克服，以便在这种更加复杂和不可预测的环境中实现有效的学习。

3.3 分析

在比较 Sarsa（一种 on-policy 学习算法）和 Q-learning（一种 off-policy 学习算法）时，我们可以看到两种方法在策略学习和探索方式上的根本差异。这些差异在 Stochastic 模式下的表现尤为明显，因为环境的不确定性和随机性对学习过程产生了显著影响。

Sarsa 作为一种 on-policy 算法，它的学习过程更加稳健和保守。在每个 Episode 以及每个 Step 中，Sarsa 都会执行 epsilon-greedy 探索策略，这意味着即使在学习过程中，它也在一定程度上保留了探索新状态的可能性。这种方法使得 Sarsa 能够更好地适应环境中的随机性，因为它的策略始终考虑了探索的元素。然而，这种保守的学习方式也可能导致收敛速度相对较慢，特别是在环境较为复杂或存在较大随机性的情况下。

另一方面，Q-learning 作为一种 off-policy 算法，更倾向于利用经验积累，寻找并学习到最优策略。这种方法在收集和利用信息方面可能更高效，因为它专注于学习可能获得最大奖励的策略，而不是当前策略下的期望奖励。这导致 Q-learning 在面对确定性环境时通常收敛更快。然而，在 Stochastic 模式下，由于环境的不确定性和随机性，Q-learning 可能表现出较大的波动性和不稳定性。这是因为在随机环境中，对未来奖励的估计更加困难，Q-learning 的这种利用导向的学习策略可能导致它对环境变化的适应性不如 Sarsa。

除了此次实验所展现的效果以外，在著名的悬崖行走曲线上，也能体现出来以上分析结果[4]————q-learning 方法是 optimal 的，但会有风险，sarsa 方法是安全的，但学习的 episode 曲线也被拉长了。

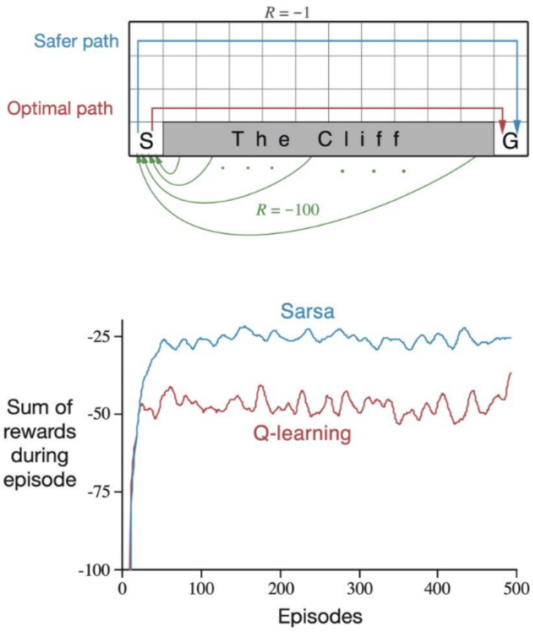

图 14 Example

综上所述，Sarsa 的学习过程由于其在策略和动作上的同时考虑，可能更为保守，导致收敛时间较长，但在随机环境中可能表现出更好的稳定性。而 Q-learning 由于其对最优策略的追求，可能在确

定性环境中表现得更优，但在面对随机环境时，其稳定性可能受到挑战。因此，在选择算法时，需要考虑到环境的特性以及算法的这些基本特征。

3.4 DQN 算法结果与分析

DQN 算法得益于它将深度学习与强化学习相结合的创新，与上两个算法有较大的区别，因此单独进行说明。下图是将所有回合以每 10 个一组进行编组，然后计算组内能够到达终点的回合的个数，并统计概率。

可以看到，算法展现了优秀的收敛性，在 1000 回合之内就能够收敛在胜率为 100 处。

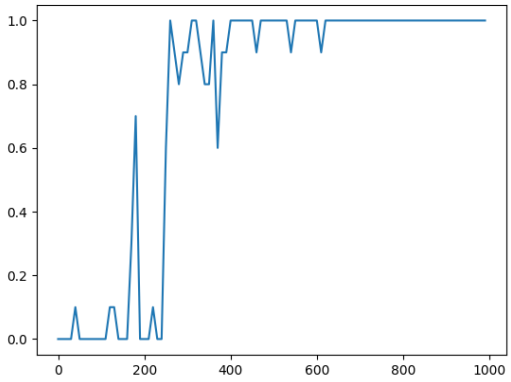

图 15 DQN-WinRate-Deterministic

当 DQN 算法应用于“冰湖”实验中，并与传统的 Q-learning 和 Sarsa 算法进行比较时，它展现出了一些独特的优势，尤其在快速达到高准确度方面。以下是与 Q-learning 和 Sarsa 相比，DQN 在冰湖实验中的几个显著优点：

1. DQN 通过经验回放机制减少数据间的时间相关性，这在冰湖这种有潜在随机性的环境中尤为重要。而传统的 Q-learning 和 Sarsa 算法可能会因为连续的状态转移而受到较大的影响。
2. DQN 利用固定 Q 目标和在线学习的方法，提高了算法的稳定性和效率。在 Q-learning 和 Sarsa 中，学习过程可能因为估计的 Q 值与目标 Q 值之间的相关性而变得不稳定，特别是在有随机性的环境中。
3. DQN 使用非线性函数逼近（通过深度网络），这使得它在学习复杂函数映射方面比传统的线性方法（如表格形式的 Q-learning 和 Sarsa）更有效。

DQN 的这些优势也伴随着更高的计算复杂度和对数据的依赖性，可能需要更多的计算资源和数据来实现其优越的性能。在 Deterministic 模式下，这些优势使得 DQN 能够更快地适应环境，学习有效的策略，并在较少的回合数内实现较高的准确度。

但是这些优势在面对 Stochastic 模式的时候却带来了灾难性的后果。

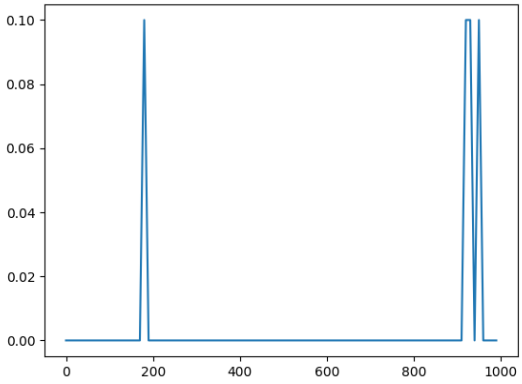

图 16 DQN-WinRate-Stochastic

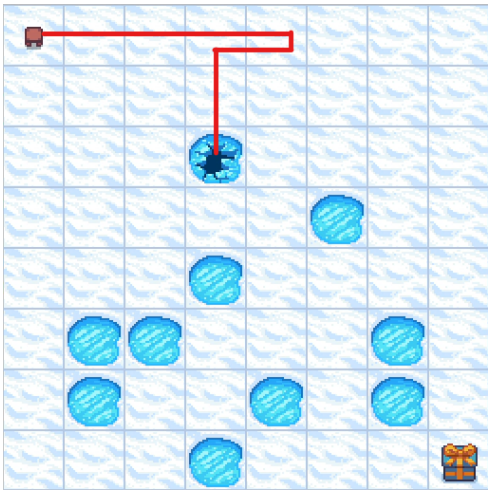

图 17 DQN-Stochastic 路径

经过分析，认为可能导致这一现象的原因如下：

1. DQN 使用的深度学习模型在面对高度随机和不确定的环境时可能难以有效学习。在 Stochastic 模式下，环境的每个状态转换都可能

包含不确定性，这使得学习过程中对状态-动作对的价值估计变得复杂和不准确。

2. 虽然经验回放机制可以减少数据间的相关性并增加学习稳定性，但在高度随机的环境中，它可能不足以克服学习过程中的挑战。过去的经验可能因环境的随机性而变得不再适用，从而降低学习效率。

3. 在固定 Q 目标方法中，目标网络的更新可能滞后于主网络，这在确定性环境中有助于稳定学习，但在随机环境中可能导致不准确的价值估计，因为环境的变化速度可能超过了目标网络更新的速度。

4. DQN 依赖于预测每个动作的潜在结果来决定其行动，但在 Stochastic 模式下，即使相同的状态和动作也可能导致不同的结果。这种不确定性使得 DQN 难以准确预测和学习最优策略。

5. DQN 的性能在很大程度上依赖于高质量的训练样本。在随机环境中，获取具有代表性和一致性的训练样本更加困难，这可能导致模型无法有效学习。

4 问题与解决

4.1 算法的效果展示与评估

在强化学习性能评估中，通常有两种主要的方法：一是计算平均得分（即 reward），这涉及在 agent 执行一定步数后，记录其获得的所有奖励值并求其平均值；二是计算平均 Q 值，即在测试性能前预先确定一定数量的状态-动作对，并在测试时计算这些确定状态-动作对的 Q 值的平均值。在我们的实验中，选择了第一种方法，即求得分的方法。

然而，在绘制 reward-epoch 曲线时，我们发现了一个问题。尽管这个曲线可以反映出随着训练轮数增加模型性能的变化趋势，但曲线的局部平稳性不佳，容易出现“抖动”。这意味着相邻 epoch 的得分往往存在较大的差距。我们分析认为，这种现象的原因在于，强化学习在每次迭代更新策略参数之后，相应的训练数据分布也发生变化。这导致训练数据的分布往往与当前的策略分布存在一定差距，并且训练数据分布的更新也存在滞后性。这可能使得算法在某些时刻对某些状态有较好的表现，而对另一些状态则表现较差。

为了解决这个问题，我们采用了滑动窗口平均的方法来平滑 reward 曲线。这种方法通过计算在特定窗口内的平均值来减少短期波动的影响，从而使得长期趋势更加明显。我们在曲线中应用了这种滑动窗口平均，这不仅使得曲线更加平滑，也更有助于观察和分析模型性能的真实趋势。

然而，由于实验条件的限制，以及强化学习训练本身耗费的时间巨大（Stochastic 模式下训练一次用时 30min），我们在进行这项改进时适当减少了试验次数。我们认为，即使在减少试验次数的情况下，通过滑动窗口平均方法改善的 reward-epoch 曲线仍能有效地反映出模型性能的变化趋势，并为我们提供更加稳定和可靠的性能评估。

5 实验总结

通过本次实验任务，我们小组成员首次深入接触并认真研究了强化学习的相关知识，获得了宝贵的经验。特别是将算法迁移到实机操作中的时候让我们学到很多。这些经验教训表明，在强化学习的应用中，仅仅追求仿真环境中的 SOTA 性能是不够的。算法的实际应用效果同样重要，特别是在处理实际物理环境中的不确定性和复杂性时。为了提高算法在真实世界中的适用性和鲁棒性，我们需要更加关注算法与实际应用环境之间的紧密结合，并在未来的工作中重视这一点。

目前，深度强化学习方法存在以下 3 个方面的局限性[5]：

1. 深度强化学习理论支撑不够。谷歌的 DeepMind 团队于 2015 年在《自然》杂志上发表的文章虽然取得了较好的应用效果，但没有证明 DQN 的收敛性，并且到目前为止在 DQN 或其他深度强化学习方法的基础上的改进工作也没有很好地解决该问题。

2. 样本采样率低。样本采样率低使得深度强化学习方法有时在实际应用中效果不佳。导致该问题的主要原因有两个：一是完成任务需要收集大量数据；二是训练过程中利用当前数据的有

用信息效率低。

3. 在连续动作空间中应用有限。目前主流的深度强化学习方法大多适用于离散动作空间，对于机器人的机械臂路径规划等连续动作空间的任务还处于初步研究阶段，理论支撑不够，因此应用十分有限。

对于强化学习在应用中遇到的问题，我们认为未来的研究应着重关注以下几个方面，以期解决现有的困境和挑战：

1. 设计有效的奖励函数：强化学习的核心在于通过最大化奖励值来寻求最优策略，而奖励函数的设计直接影响到所得策略的优劣。目前，奖励函数多由专家根据其专业知识设计，但随着路径规划领域应用环境的日益复杂化，传统的奖励函数设计方法可能不再适用。一些学者提出了元学习（Meta Learning）等新方法，旨在使智能体在面对环境或任务变化时，能从一系列合理的策略中不断优化其奖励函数。因此，设计有效且适应性强的奖励函数是未来研究的关键方向之一。

2. 解决探索与利用的困境：在强化学习中，探索意味着不断尝试新的环境信息以获得更高的奖励值，防止陷入局部最优；利用则指依据已学习到的信息选择预期奖励最高的行为。探索与利用之间的平衡是强化学习中的一个核心问题。目前，E-greedy 算法是常用的一种平衡方法，它让智能体以一定的概率随机探索或利用信息。虽然 E-greedy 算法简单易实现，但其随机探索效率低下，因此如何更高效地解决探索与利用的困境需要进一步研究。

3. 研究强化学习与其他方法的结合：各种强化学习方法在应用于路径规划时都有其局限性。为了克服单一方法的不足，结合不同方法的优势可能是一个有效的策略。例如，传统的路径规划算法、图形学算法、智能仿生学算法与强化学习算法的结合，通过各自的长处相互补充，有望产生性能更佳的综合方法。

当前，基于 DQN 的深度强化学习模型已相对成熟，策略梯度方法得到广泛应用，同时元学习等领域的新算法也不断应用于深度强化学习。尽管如此，作为机器学习的新兴领域，深度强化学习仍在不断发展之中，许多问题有待进一步探索和研究。我们团队也将持续学习相关知识，努力在这一前沿领域进行深入探索。

参考文献

[1] CSDN. Q-learning-introduction[EB/OL]. https://blog.csdn.net/shoppingend/article/details/124291112.

[2] AMAZON. what-is-srasa[EB/OL]. https://www.amazonaws.cn/knowledge/what-is-sarsa/.

[3] CSDN. Dqn-introduction[EB/OL]. https://blog.csdn.net/Zhang_0702_China/article/details/123423637.

[4] RUOYU. 强化学习中 sarsa 算法是不是比 q-learning 算法收敛速度更慢？[EB/OL]. https://www.zhihu.com/question/268461866.

[5] ENDERFGA. RI[EB/OL]. https://enderfga.cn/2022/11/22/RL/.

2023-EXP-Frozen_Lake

代码运行指南

脱机模拟

代码存放在offline文件夹中

q_learning & Sarsa

环境配置：

新建conda环境：`python==3.7`，然后在conda环境中执行（pip也需要使用当前conda环境的pip）

```
pip install tensorflow==1.15
pip install gym==0.23.1
pip install pygame
pip install tqdm
pip install matplotlib
conda install -c conda-forge gcc  # 在无英伟达显卡的设备上运行的时候需要执行这一指令
```

在`q_learning+Sarsa`文件夹下启动Terminal，`conda activate xxx`以后

执行以下代码开始训练或进行演示。

```
python -m q_learning.train
python -m q_learning.load
```

```
python -m Sarsa.train
python -m Sarsa.load
```

DQN

新建conda环境：`python==3.8`，然后在conda环境中执行（pip也需要使用当前conda环境的pip）。

```
pip install tensorflow==2.10
pip install gym==0.23.1
pip install pygame
pip install tqdm
pip install matplotlib
```

进入DQN文件夹中，激活环境后执行`python DQN.py`即可开始训练，训练结束以后自动进行仿真实验。

代码说明文档

Q-learning

由于q-learning算法和sarsa算法仅仅在下一步动作的选择策略上有所区别，而其他框架一致，我们这里主要框架仅以q-learning模块为例

执行顺序：

执行 train.py模块即可完成全部的过程；

修改参数：

- 训练轮数：直接在train.py的train函数中修改
- 地图尺寸及模式：地图模块进行修改
- 修改地图尺寸后，需要修改tabular_q_agent.py模块test模块中的taction模块，有注释，可以参考

模型初始化

utils.py：

```
import random
def set_global_seeds(i):
    try:
        import tensorflow as tf
    except ImportError:
        pass
    else:
        tf.set_random_seed(i)
    try:
        import numpy as np
    except ImportError:
        pass
    else:
        np.random.seed(i)
    random.seed(i)
```

init.py：

```
from .utils import set_global_seeds

set_global_seeds(0)
```

通过设置随机种子保证了在i值一定的时候随机生成的数据一致，保证了全局实验的可复现。

地图模块

map.py：

```
custom_map = [
'SFFHF',
'HFHFF',
'HFFFH',
'HHHFH',
'HFFFG'
]
custom_map=None
map_config={'desc':custom_map,'map_name':'8x8','is_slippery':False}
```

custom_map实现自定义地图的功能，map_config设置一些无参数传入状态下地图参数的默认值

算法模块

tabular_q_agent.py:

我们仅介绍算法的一些核心模块，保证能够最快的复现实验过程

导入包及功能：

```
from collections import defaultdict
# 一个有用的数据结构，dict的子类，用于创建默认值为零的字典

import functools
# 一个重要的功能是提供了 functools.wraps 装饰器，用于更新被装饰函数的元数据，以便更好地保留原函数的信息。

import pickle
# pickle导入导出

import numpy as np
import time

from gym.spaces import discrete
# gym 提供了 gym.spaces 模块来处理不同类型的状态空间和动作空间。
# gym.spaces 模块中的 discrete 类是用于定义离散型空间的一种，它表示一个有限的整数集合，代表了离散的状态或动作空间。
# 可以用数来表示离散空间的动作范围

class UnsupportedSpace(Exception):
    pass
```

接下来介绍算法类主要的四个模块

init:实现初始化功能

if not isinstance（A，B）检查A是否是B的类型，确保观察到的空间和行为空间都是离散的空间

```
    if not isinstance(observation_space, discrete.Discrete):
        raise UnsupportedSpace('Observation space {} incompatible with {}. (Only
supports Discrete observation spaces.)'.format(observation_space, self))
```

存储一些重要的信息，初始Q表的平均值，方差，学习率，探索率，折扣率，最大探索步数

```
    self.config = {
        "init_mean": 0.0,      # Initialize Q values with this mean
        "init_std": 0.0,       # Initialize Q values with this standard deviation
        "learning_rate": 0.5,
        "eps": 0.05,            # Epsilon in epsilon greedy policies
        "discount": 0.99,
        "n_iter": 10000}        # Number of iterations
```

创建generate_zeros的部分应用版本，将其中的参数n值具体为某个数

```
    self.q = defaultdict(functools.partial(generate_zeros, n=self.action_n))
```

act: 采取行为时有概率进行探索，保证了算法不至于陷入鼠目寸光陷阱

eps: 探索率，探索失败时采取贪心策略

observation: 当前的观测空间

```
    # epsilon greedy.
    action = np.argmax(self.q[observation]) if np.random.random() > eps else
self.action_space.sample()
```

learn:进行多轮训练更新Q表的过程

obs:当前状态 obs2：下一步状态（位置）

设置回报 # Get negative reward every step if reward == 0: reward = -0.005

```
        # if agent sucked at same position, punish it
        if obs == obs2:
            reward = -0.01

        # if agent fill to hole then die, punish it
        if done and not reward:
            reward = -1
```

```
        future = 0.0
        if not done:
            future = np.max(self.q[obs2])
        # 没有到下一步就会采取贪心策略
```

Q-learning算法的更新策略

```
        self.q[obs][action] = (1 - learning_rate) * self.q[obs][action] +
learning_rate * (reward + self.config["discount"] * future)

        rAll += reward
        step_count += 1
        # 记录最终的回报和步数
```

test模块：利用我们的Q表控制飞机飞跃冰湖的过程

```
    # 这里为了更好的保证飞机的运行，不至于驶出地图外，我们增加了一些限制，
    # 但事实上，由于算法正确，完全可以去掉这一部分限制
    maplocationX=0
    maplocationY=0
```

飞机的起飞动作应该在后续命令动作之前

```
    self.st.send().takeoff(50)
    time.sleep(3)
```

贪心策略确定下一步命令

```
        action = self.act(obs, eps=0)
        obs2, reward, done, _ = env.step(action)
```

这里为了保证在stochastic模式下也能够正常运行，我们通过人物实际的位置来确定实际执行命令，并且依此给飞机发送命令

```
        dis=obs2-pos
        pos=obs2
        # dis后面的限制值应该根据地图大小进行修改，如果为4x4则改成dis==-4，下同
        if dis==-8:
            taction=3
        elif dis==8:
```

```
            taction=1
        elif dis==-1:
            taction=0
        elif dis==1:
            taction=2
        else:
            taction=-1
```

给飞机输送命令

```
            if taction == 3:  # up
                if maplocationY > 0:
                    self.st.send().forward(50)
                    time.sleep(3)
                    print("tag_id==",end="")
                    print(self.st.vision_sensor_info().tag_id)
                    time.sleep(1)
                    maplocationY-=1
                else:
                    maplocationY=0
                    pass
```

到达降落

```
        if done:
            self.st.send().land()
            time.sleep(3)
            # break
            if not reward:
                return 0,t+1
            return 1,t+1
```

train模块：执行指令

导入包：

```
from tqdm import tqdm
# tqdm模块的作用是将我们的进程进行实时的展示

from terminaltables import AsciiTable
# 这一个主要用来美化我们的输出结果，创建各种表格，完成结果输出的美化

from .tabular_q_agent import TabularQAgent
# 最最主要的算法模块
```

```
from .map import map_config
# 记录了我们需要使用的地图的信息
```

为了保证Q-learning算法更加有效，对学习率等参数做衰减操作

```
def exponential_decay(starter_learning_rate, global_step, decay_step,
                  decay_rate, mini_value=0.0):
    decayed_learning_rate = starter_learning_rate *
math.pow(decay_rate,math.floor(global_step / decay_step))

    return decayed_learning_rate if decayed_learning_rate > mini_value else
mini_value

    # 这里是后面训练过程中的操作
    learning_rate = exponential_decay(0.9, episode, 1000, 0.99)
    eps_rate = exponential_decay(1.0, episode, 1000, 0.97, 0.001)
```

保存我们的参数，保证结果可复现：

```
def save_rewards_as_pickle(rewards, filename='q_learning-rewards.pkl'):
with open(filename, 'wb') as file:
    pickle.dump(rewards, file)
```

结果展示的美化：

```
table_header = ['Episode', 'learning_rate', 'eps_rate', 'reward', 'step']
rewards = []
table_data = [table_header]

table_data.append([episode, round(learning_rate,3), round(eps_rate,3),
round(all_reward,3), step_count])
table = AsciiTable(table_data)
# 这里语句的作用就是直观显示每一轮训练过程的数据
tqdm.write(table.table)
# 实时化训练过程
```

通过和归一化的矩阵做卷积函数实现回报的归一化，并且将结果展示出来，也就是我们看到的那张收敛图

```
smoothed_data = np.convolve(rewards, np.ones(window_size)/window_size,
mode='valid')
```

其次就是调用具体的算法函数来实现过程，不再赘述

Sarsa算法

tabular_q_agent.py的learn函数中：

```
future = self.q[obs2][action2]
# 和Q-learning算法就只有这一步区别，下一步动作的选取上的区别
```

DQN算法

执行过程

- env：设置地图
- dqn.run(times):times表示训练轮数

导入包：

```
import tensorflow.compat.v1 as tf
# 现有tensorflow版本都是2.x版本，这里提供与 TensorFlow 1.x 版本兼容的接口
import numpy as np
import gym
import matplotlib.pyplot as plt
```

具体网络DQN的搭建：

init初始化：

```
self.nstate=nstate   # 状态空间的维度
self.naction=naction  # 动作空间的维度
self.sess = tf.Session()  # tensorflow会话，用于执行计算图
self.memcnt=0      # 记录回放缓冲区中当然存储的数据数量
self.BATCH_SIZE = 64  # 每次从缓冲区中采样量的大小
self.LR = 0.0012                    # learning rate
self.EPSILON = 0.92                 # greedy policy
self.GAMMA = 0.9999                 # reward discount
self.MEM_CAP = 2000     # 回放缓冲区大小
self.mem= np.zeros((self.MEM_CAP, self.nstate * 2 + 2))     # initialize memory
self.updataT=150       # 更新目标网络的频率
self.built_net()        # 方法用于构建神经网络
```

搭建网络： 评估网络和目标网络

定义状态、动作、奖励、下一状态的占位符，用于输入各个数据

```
self.s = tf.placeholder(tf.float64, [None,self.nstate])
self.a = tf.placeholder(tf.int32, [None,])
self.r = tf.placeholder(tf.float64, [None,])
self.s_ = tf.placeholder(tf.float64, [None,self.nstate])
```

定义两个网络

训练网络：

l_eval: 代表评估网络的隐藏层，具有10个神经元，激活函数为 ReLU。

self.q: 代表 Q-value 的输出层，其神经元数量为 self.naction，即动作的数量。这是代理根据当前状态估计的 Q-value。

```
with tf.variable_scope('q'):                                    # evaluation network
l_eval = tf.layers.dense(self.s, 10, tf.nn.relu,
kernel_initializer=tf.random_normal_initializer(0, 0.1))
self.q = tf.layers.dense(l_eval, self.naction,
kernel_initializer=tf.random_normal_initializer(0, 0.1))
```

目标网络：

l_target: 代表目标网络的隐藏层，具有10个神经元，激活函数为 ReLU。

q_next: 代表目标网络的输出层，其神经元数量为 self.naction。目标网络的参数在训练过程中不会更新（trainable=False）。

```
 with tf.variable_scope('q_next'):                # target network, not to train
    l_target = tf.layers.dense(self.s_, 10, tf.nn.relu, trainable=False)
    q_next = tf.layers.dense(l_target, self.naction, trainable=False)
```

计算目标的q-value和当前估计的q-value

```
    q_target = self.r + self.GAMMA * tf.reduce_max(q_next, axis=1)    #q_next:
shape=(None, naction),
    # 计算当前估计的q-value
    a_index=tf.stack([tf.range(self.BATCH_SIZE,dtype=tf.int32),self.a],axis=1)
    q_eval=tf.gather_nd(params=self.q,indices=a_index)
```

定义损失、优化器、初始化

```
loss=tf.losses.mean_squared_error(q_target,q_eval)
# 损失函数
```

```
self.train=tf.train.AdamOptimizer(self.LR).minimize(loss)
#  q现实target_net- Q估计，定义优化操作
self.sess.run(tf.global_variables_initializer())
# 初始化变量，保证模型可以正确运行
```

greedy策略选择动作

```
if  np.random.uniform(0.0,1.0)<self.EPSILON:
    action=np.argmax( self.sess.run(self.q,feed_dict={self.s:fs}))
else:
    action=np.random.randint(0,self.naction)
```

learn

每次训练前，先判断是否需要对目标网络进行更新，判断完成后再从回放缓冲区随机选择一批样本，利用这批样本对评估网络进行一次优化操作

```
def learn(self):
    if(self.memcnt%self.updataT==0):
        t_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
scope='q_next')
        e_params = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='q')
        self.sess.run([tf.assign(t, e) for t, e in zip(t_params, e_params)])
    rand_indexs=np.random.choice(self.MEM_CAP,self.BATCH_SIZE,replace=False)
    temp=self.mem[rand_indexs]
    bs = temp[:,0:self.nstate]#.reshape(self.BATCH_SIZE,NSTATUS)
    ba = temp[:,self.nstate]
    br = temp[:,self.nstate+1]
    bs_ = temp[:,self.nstate+2:]#.reshape(self.BATCH_SIZE,NSTATUS)
    self.sess.run(self.train, feed_dict=
{self.s:bs,self.a:ba,self.r:br,self.s_:bs_})
```

存储数据进入回放缓冲区

```
def storeExp(self,s,a,r,s_):
fs = np.zeros(self.nstate)
fs[s] = 1.0                         # ONE HOT
fs_ = np.zeros(self.nstate)
fs_[s_] = 1.0                             # ONE HOT
self.mem[self.memcnt%self.MEM_CAP]=np.hstack([fs,a,r,fs_])
self.memcnt+=1
# one-hot编码后，存储到回放缓冲区，采用循环队列的模式，确保不会超过回放缓冲区的容量
```

展示结果

```
def show(self):
    print("show")
    obs = env.reset()
    env.render('human')
    for t in range(10000):
        env.render('human')
        action = dqn.choose_action(obs)
        obs2, reward, done, _ = env.step(action)
        env.render('human')
        if done:
            break
        obs = obs2
```

集成所有模块运行

记录用数据

```
    cnt_win = 0 # 记录最近50次中完成任务的次数
    winrate_recorder = 0 # 记录最近10回合中成功完成任务的次数
    all_r=0.0  # 记录所有回合的累计奖励
    win_rate=[]
```

每次训练后存储

```
a=self.choose_action(s)
s_,r,done,_=env.step(a)
all_r+=r
self.storeExp(s,a,r,s_)
```

经验池满则网络进行一次学习：

```
 if(self.memcnt>self.MEM_CAP):
    self.learn()  # 经验池满，则进行一次学习
```

更新记录用数据

```
if(done):
    if(s_==self.nstate-1):
        cnt_win+=1.0
        winrate_recorder+=1.0
```

根据最近50次的任务完成率对贪心策略的执行率做出调整，越大越不可能执行随机动作

```
if (cnt_win / 50 > 0.4):
        self.EPSILON += 0.01
elif (cnt_win / 50 > 0.2):
        self.EPSILON += 0.005
elif (cnt_win / 50 > 0.1):
        self.EPSILON += 0.003
elif (cnt_win / 50 > 0.05):
        self.EPSILON += 0.001
```

绘制结果曲线

```
x_axis = [i * 10 for i in range(len(win_rate))]
plt.plot(x_axis, win_rate)
plt.show()
# 绘制训练过程中平均胜率的曲线
```