课程作业5：强化学习实验报告

摘　要： 本报告详细记录了在强化学习课程作业中的实验过程和成果。实验主要包括价值迭代、Q 学习、epsilon 贪心策略、以及近似 Q 学习等方法的实现和测试。通过在格子世界和吃豆人游戏环境中对智能体进行训练和测试，我们深入探索了强化学习算法的理论基础和实际应用。实验结果显示，所实现的算法在不同的测试环境中均表现出良好的性能，有效地解决了一系列决策问题。这些实验不仅加强了对强化学习理论的理解，也展示了其在复杂环境中的应用潜力。

关键词： 强化学习；价值迭代；Q 学习；epsilon 贪心策略；近似 Q 学习；吃豆人；智能体。

1　引言

在当今信息时代，强化学习作为人工智能领域的一个重要分支，正在迅速发展并在各个领域展现出巨大的应用潜力。本课程作业专注于强化学习的基本理论和应用实践，旨在通过具体的编程任务深入探索和理解其核心算法和实际应用。本实验报告详细记录了包括价值迭代、策略、Q 学习、epsilon 贪心策略和近似 Q 学习等多种强化学习方法的实现过程与实验结果。通过这些任务，不仅加深了对强化学习理论的理解，也锻炼了将理论应用到实际问题中的能力。报告的最终目的是展现实验过程中的学习成果，并为今后在强化学习及相关领域的研究提供参考和启示。

2　实验内容

2.1　任务1：价值迭代（Value Iteration）

2.1.1　代码实现

在任务 1 中，我们实现了一个名为 `ValueIterationAgent` 的类，它是 `ValueEstimationAgent` 的子类。这个类的目的是通过价值迭代方法来解决强化学习中的决策问题。我们首先初始化了一些基本属性，包括马尔可夫决策过程（MDP），折扣因子和迭代次数。随后，我们实现了 `runValueIteration` 方法，该方法通过迭代计算每个状态的价值。在每次迭代中，我们对所有状态进行遍历，计算并更新它们的价值。此外，我们实现了几个辅助方法，例如 `computeQValueFromValues` 和 `computeActionFromValues`，这些方法分别用于计算 Q 值、决定最佳行动。

以下是 `runValueIteration` 、 `computeQValueFromValues`、 `computeActionFromValues` 方法的具体实现：

1. `runValueIteration` 方法：

这个方法是实现值迭代算法的主要部分。值迭代是一种用于求解马尔可夫决策过程（MDP）的算法。在每次迭代中，它会更新每个状态的值，这个值是基于当前状态可能的所有动作的最大预期回报。

首先，代码进行了一次循环，循环次数为预设的迭代次数。在每次迭代中，它创建了一个临时的计数器 `temp`，用于存储每个状态的新值。

然后，代码遍历了 MDP 中的所有状态。对于每个状态，如果它是一个终止状态（即没有未来的奖励），那么它的值就被设为 0。否则，代码会计算该状态下所有可能动作的 Q 值，并取最大的 Q 值作为该状态的值。这里的 Q 值是通过 `computeQValueFromValues` 函数计算的，该函数根据状态转移的概率和奖励，以及下一个状态的值来计算 Q 值。

最后，当所有状态的值都计算完毕后，这些值会被赋给 `self.values`，以便在下一次迭代中使用。

这个过程会一直重复，直到达到预设的迭代次数。最终，`self.values` 将包含了 MDP 中每个状态的最优值。

2. `computeQValueFromValues` 方法：

这个方法是计算 Q 值的函数，Q 值是强化学习中的一个重要概念，表示在某个状态下执行某个动作的预期回报。

首先，函数初始化了 `qValue` 为 0，然后遍历了从当前状态 `state` 执行动作 `action` 可能到达的所有下一个状态及其转移概率。这些信息是通过调用 `getTransitionStatesAndProbs` 函数获得的。

对于每一个可能的下一个状态 `nextState` 及其转移概率 `prob`，函数计算了从 `state` 执行 `action` 到达 `nextState` 的预期回报。这个预期回报是由转移概率、即时奖励和折扣后的下一个状态的值的乘积得到的。即时奖励是通过调用 `getReward` 函数获得的，下一个状态的值则是从 `self.values` 中查找的。这个预期回报被累加到 `qValue` 中。

最后，函数返回计算得到的 `qValue`。这个值表示在当前状态下执行某个动作的预期回报，将被用于值迭代算法中更新状态的值。

3. `computeActionFromValues` 方法：

这个方法是用于从当前存储在 `self.values` 中的值函数计算给定状态下最佳动作的函数。这个函数是强化学习中策略（policy）的一部分，策略是指在给定状态下选择最佳动作的规则。

首先，函数初始化了 `bestAction` 为 `None`，并将 `bestValue` 设置为负无穷大。这是为了在后续的循环中找到最大的 Q 值及其对应的动作。

然后，函数遍历了在给定状态 `state` 下可能的所有动作。这些动作是通过调用 `getPossibleActions` 函数获得的。

对于每一个可能的动作，函数计算了在当前状态下执行该动作的 Q 值。这个 Q 值是通过调用 `computeQValueFromValues` 函数计算的，该函数根据状态转移的概率和奖励，以及下一个状态的值来计算 Q 值。

如果计算得到的 Q 值大于当前的 `bestValue`，那么就更新 `bestValue` 和 `bestAction`。这样，循环结束后，`bestAction` 就是 Q 值最大的动作，也就是在当前状态下的最佳动作。

最后，函数返回 `bestAction`。如果没有可行的动作（例如在终止状态），那么 `bestAction` 将保持为 `None`。

2.1.2 实验结果

```
Question q1
===========

*** PASS: test_cases\q1\1-tinygrid.test
*** PASS: test_cases\q1\2-tinygrid-noisy.test
*** PASS: test_cases\q1\3-bridge.test
*** PASS: test_cases\q1\4-discountgrid.test

### Question q1: 6/6 ###


Finished at 21:06:16

Provisional grades
==================
Question q1: 6/6
```

```
------------------
Total: 6/6
```

由此可见，代码通过了所有的测试用例，我们的价值迭代算法在多种测试条件下均能稳定工作，充分验证了其有效性和鲁棒性。这些结果为总分6分中的满分6分提供了支持，表明实验完全符合预期标准。

2.2 任务2：策略（Policies）

2.2.1 代码实现

在任务 2 中，我们探索了不同的策略（Policies）配置对强化学习行为的影响。代码实现包括五个不同的函数，每个函数针对特定的策略设定返回了不同的参数配置：

1. `question2a`：优先选择近处的出口（奖励+1），冒着掉落悬崖的风险（惩罚-10）。这里使用了较低的折扣因子（0.1）和零噪声。
2. `question2b`：同样优先选择近处的出口，但尽量避免掉落悬崖。增加了一定的噪声（0.1），以模拟避免风险的行为。
3. `question2c`：优先选择远处的出口（奖励+10），同样冒着掉落悬崖的风险。使用较高的折扣因子（0.9）和零噪声。
4. `question2d`：优先选择远处的出口，同时避免悬崖。增加了更高的噪声（0.2）。
5. `question2e`：避免所有出口和悬崖，使得情节永远不会结束。使用了最大的折扣因子（1）、一定的噪声（0.1）和非常高的生存奖励（100），以模拟这种策略。

这些函数展示了如何通过调整强化学习算法的参数来影响智能体的行为策略。

2.2.2 实验结果

```
Question q2
===========

*** PASS: test_cases\q2\1-question-2.1.test
*** PASS: test_cases\q2\2-question-2.2.test
*** PASS: test_cases\q2\3-question-2.3.test
*** PASS: test_cases\q2\4-question-2.4.test
*** PASS: test_cases\q2\5-question-2.5.test

### Question q2: 5/5 ###


Finished at 21:13:00

Provisional grades
==================
Question q2: 5/5
------------------
Total: 5/5
```

由此可见，代码通过了所有的测试用例，验证了我们在策略设计中的考量，展示了智能体在不同环境下的适应能力和有效性。

2.3 任务3：Q 学习（Q Learning）

2.3.1 代码实现

在任务 3 中，我们实现了一个 `QLearningAgent` 类，该类通过 Q 学习方法实现强化学习。该类初始化时建立了一个 Q 值表，用于存储各状态-动作对的 Q 值。主要方法包括：

1. `getQValue` 方法：

   该函数返回给定状态 `state` 和动作 `action` 的 Q 值。

   在这个函数中，Q 值是通过查找字典 `self.qValues` 来获取的，其中键是一个由状态和动作组成的元组。如果我们从未见过这个状态，或者对应的 Q 节点值不存在，那么应该返回 0.0。否则，返回对应的 Q 值。

2. `computeValueFromQValues` 方法：

   该函数返回给定状态 `state` 下所有可能动作的最大 Q 值。

   首先，函数初始化 `maxValue` 为负无穷大。这是为了在后续的循环中找到最大的 Q 值。

   然后，函数遍历了在给定状态 `state` 下可能的所有动作。这些动作是通过调用 `getLegalActions` 函数获得的。

   对于每一个可能的动作，函数获取了在当前状态下执行该动作的 Q 值。这个 Q 值是通过调用 `getQValue` 函数获取的。

   如果计算得到的 Q 值大于当前的 `maxValue`，那么就更新 `maxValue`。这样，循环结束后，`maxValue` 就是所有动作中 Q 值最大的值，也就是当前状态的值。

   最后，函数返回 `maxValue`。如果没有可行的动作（例如在终止状态），那么 `maxValue` 将保持为 0.0。

3. `computeActionFromQValues` 方法：

   该函数计算在给定状态 `state` 下应该采取的最佳动作。

   首先，函数初始化 `maxValue` 为负无穷大，`bestAction` 为 None。这是为了在后续的循环中找到 Q 值最大的动作。

   然后，函数遍历了在给定状态 `state` 下可能的所有动作。这些动作是通过调用 `getLegalActions` 函数获得的。

   对于每一个可能的动作，函数获取了在当前状态下执行该动作的 Q 值。这个 Q 值是通过调用 `getQValue` 函数获取的。

   如果计算得到的 Q 值大于当前的 `maxValue`，那么就更新 `maxValue` 和 `bestAction`。这样，循环结束后，`bestAction` 就是 Q 值最大的动作，也就是应该采取的最佳动作。

   最后，函数返回 `bestAction`。如果没有可行的动作（例如在终止状态），那么 `bestAction` 将保持为 None。

4. `update` 方法：

   该函数根据观察到的状态转换和奖励来更新 Q 值表。

   在函数中，首先计算了一个名为 `sample` 的值，这个值是当前奖励 `reward` 加上折扣因子 `self.discount` 乘以 `nextState` 的值函数。这个值函数是通过调用 `computeValueFromQValues` 函数计算的。

然后，函数更新了在当前状态 `state` 下执行动作 `action` 的 Q 值。新的 Q 值是原来的 Q 值乘以 `(1 - self.alpha)` 加上 `sample` 乘以 `self.alpha`。这里的 `self.alpha` 是学习率，用于控制新的观察对 Q 值的影响程度。

2.3.2 实验结果

```
Question q3
===========

*** PASS: test_cases\q3\1-tinygrid.test
*** PASS: test_cases\q3\2-tinygrid-noisy.test
*** PASS: test_cases\q3\3-bridge.test
*** PASS: test_cases\q3\4-discountgrid.test

### Question q3: 6/6 ###


Finished at 21:28:54

Provisional grades
==================
Question q3: 6/6
------------------
Total: 6/6
```

由此可见，代码通过了所有的测试用例，代码在实现Q-Learning Agent时，正确地计算了值函数和动作函数，能够根据当前状态选择最佳的动作，并且能够根据给定的奖励和下一个状态更新Q值。

2.4 任务4：epsilon 贪心策略（Epsilon Greedy）

2.4.1 代码实现

在任务4的代码实现中，我们应用了 epsilon 贪心策略（Epsilon Greedy）来决定智能体在给定状态下的行动。这个方法基于两种行为：以一定概率（epsilon）随机选择动作，或者选择当前估计为最佳的动作。代码首先获取当前状态下的所有合法动作。如果没有合法动作，则返回 None。

- 如果有合法动作，使用 `util.flipCoin(self.epsilon)` 来决定是随机选择动作还是选择最佳动作。
- 如果决定随机选择，则使用 `random.choice(legalActions)` 从合法动作中随机选取一个；否则，使用 `computeActionFromQValues(state)` 方法选择当前最佳动作。

这种策略平衡了探索（exploration）和利用（exploitation）的需要。

2.4.2 实验结果

```
Question q4
===========

*** PASS: test_cases\q4\1-tinygrid.test
*** PASS: test_cases\q4\2-tinygrid-noisy.test
*** PASS: test_cases\q4\3-bridge.test
*** PASS: test_cases\q4\4-discountgrid.test

### Question q4: 2/2 ###
```

```
Finished at 21:33:45

Provisional grades
==================
Question q4: 2/2
------------------
Total: 2/2
```

由此可见，我们的 epsilon 贪心策略成功通过了所有测试案例，证明了其有效性和适用性。

2.5 任务5：Q 学习与吃豆人（Q-Learning and Pacman）

2.5.1 实验结果

```
Question q5
===========
### Question q5: 2/2 ###


Finished at 21:39:16

Provisional grades
==================
Question q5: 2/2
------------------
Total: 2/2
```

吃豆人智能体在2000次训练过程中表现出明显的进步。初始阶段的平均奖励较低，但随着训练的深入，平均奖励逐渐增加，显示出智能体的学习能力。训练完成后，在100次测试中，智能体展现出极高的胜率和平均得分，达到了500.6分。这一结果表明经过充分训练的 Q 学习算法在吃豆人游戏中极为有效，能够在各种场景下取得优异成绩。智能体在此任务中的满分表现进一步验证了Q学习策略的有效性。

2.6 任务6：近似 Q 学习（Approximate Q-Learning）

2.6.1 代码实现

在任务6中，我们实现了一个近似 Q 学习智能体（Approximate Q-Learning Agent）。此智能体的目的是学习状态特征的权重，其中可能有多个状态共享相同的特征。主要方法包括：

1. `getQValue` 方法：

   该函数通过计算权重和特征向量的点积来计算 Q 值。

   首先，函数初始化 `qValue` 为 0.0。这是为了在后续的循环中累加每个特征的权重乘以特征值。

   然后，函数获取了当前状态和动作的特征向量。这个特征向量是通过调用 `self.featExtractor.getFeatures` 函数获得的。

   接着，函数遍历了特征向量中的每一个特征。对于每一个特征，函数获取了该特征的权重 `self.weights[feature]`，并将其乘以特征值 `features[feature]`，然后加到 `qValue` 上。

   最后，函数返回 `qValue`，即权重和特征向量的点积，也就是 Q 值。

2. `update` 方法：

该函数根据从状态转换和奖励中观察到的差异来更新权重。

在函数中，首先计算了一个名为 `difference` 的值，这个值是当前奖励 `reward` 加上折扣因子 `self.discount` 乘以 `nextState` 的值函数，然后减去当前状态和动作的 Q 值。这个值函数是通过调用 `computeValueFromQValues` 函数计算的，而 Q 值是通过调用 `getQValue` 函数获取的。

然后，函数获取了当前状态和动作的特征向量。这个特征向量是通过调用 `self.featExtractor.getFeatures` 函数获得的。

接着，函数遍历了特征向量中的每一个特征。对于每一个特征，函数更新了该特征的权重，新的权重是原来的权重加上学习率 `self.alpha` 乘以 `difference` 和特征值 `features[feature]`。

2.6.2 实验结果

```
Question q6
===========

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test
*** PASS: test_cases\q6\5-coord-extractor.test

### Question q6: 4/4 ###


Finished at 21:47:08

Provisional grades
==================
Question q6: 4/4
------------------
Total: 4/4
```

由此可见，代码通过了所有的测试用例，证明了近似 Q 学习策略的有效性和适用性，特别是在处理具有大量状态的复杂环境时。这种方法通过学习状态特征的权重，而不是单独为每个状态-动作对存储 Q 值，从而在复杂环境中提高了效率和可扩展性。

3 总结

在本次课程作业中，我们成功实现了价值迭代和 Q 学习，并在格子世界环境中对所实现的智能体进行了测试。这些测试进一步扩展到了更复杂的仿真环境，如仿真机器人控制器（Crawler）和吃豆人游戏。

1. **价值迭代和策略实现**：在任务1和任务2中，我们专注于价值迭代和不同策略的实现。我们的算法在基础和噪声网格测试中均表现良好，展现出了强化学习算法在处理简单至复杂决策问题时的有效性。

2. **Q 学习和epsilon 贪心策略**：在任务3和任务4中，我们通过实现 Q 学习算法和epsilon 贪心策略进一步加深了对强化学习的理解。智能体在不同的测试环境中均获得了满意的成绩，显示了算法的鲁棒性和灵活性。

3. **吃豆人游戏中的应用**：任务5和任务6将强化学习算法应用于吃豆人游戏。我们首先在游戏环境中应用了 Q 学习，然后进一步探索了近似 Q 学习。吃豆人智能体在训练期间表现出了明显的进步，最终在测试阶段取得了高分，这展示了强化学习算法在复杂、动态环境中的有效性。

总体来看，这些实验不仅加深了我们对于强化学习理论的理解，也提升了我们将理论应用于实际问题的能力。通过对不同算法的实现和测试，我们更加清楚地认识到强化学习在解决实际问题中的潜力和应用广度。