**Introduction to Artificial Intelligence**

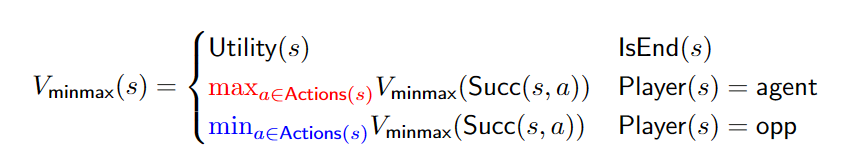
* **HW3**

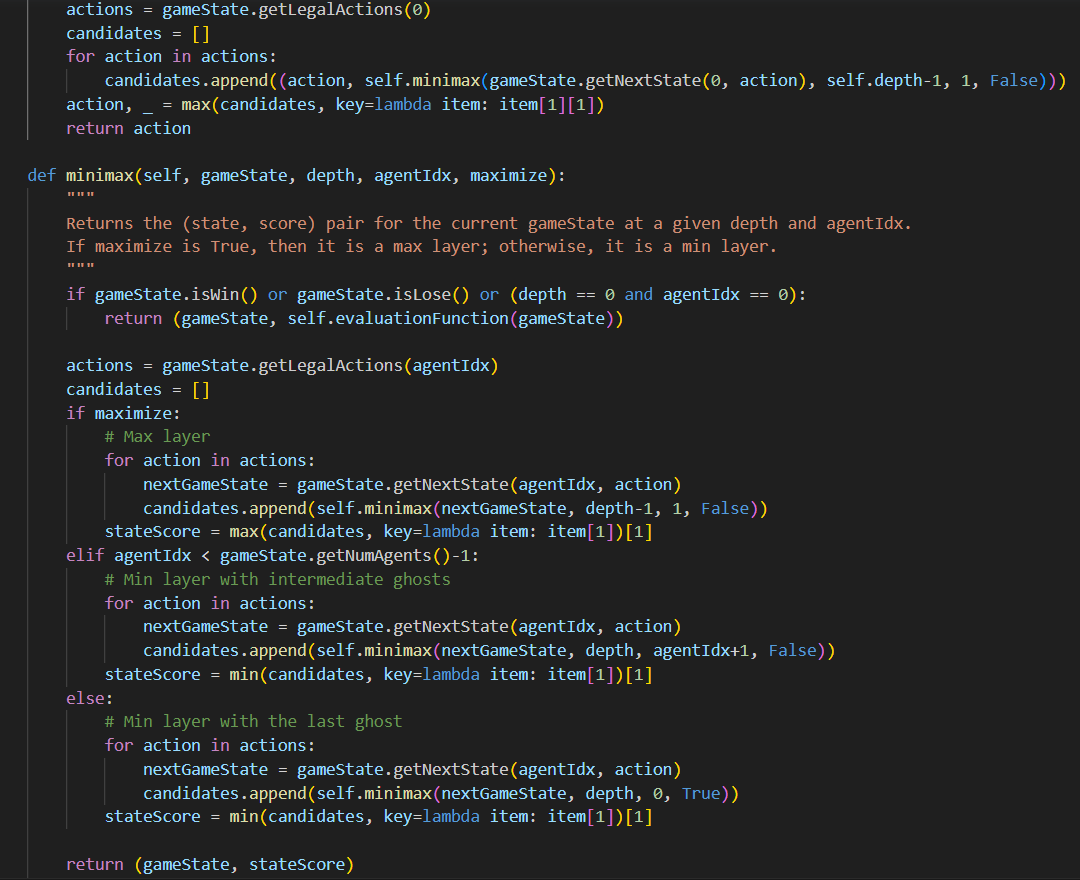
109350008 張詠哲

* **Part 1**

**1.1 MinimaxAgent**

這部分要實作Minimax演算法，用意是為了最大化自己的收益且最小化對手的收益，但因為也要假設對方是智慧體(Agent)，會最小化我們的收益。因此會考慮到執行下一步後對手會如何行動，並再從中找出收益最大的動作，其表達式如下:





getAction會對所有legalactions執行Minimax演算法以獲得該動作對應的遊戲狀態和分數。然後從這些候選動作中選擇分數最高的動作，並將其返回。

而自訂的minimax函式會獲取當前狀態下的所有legalactions。然根據當前層的類型(max layer 或 min layer)以遞歸的方式，對每個動作執行Minimax以獲得候選的(狀態, 分數)。如果是max layer，則找出最高分數的候選狀態和分數。如果是min layer，表示為GhostAgent，則找出最低分數的候選狀態和分數(會把最後一個GhostAgent拉出來寫是因為當執行到最後一個鬼後，下一個就是max layer，他的index是0)。最後當為終局或是深度被減到為0後，函數返回當前遊戲狀態和最終的狀態分數。

**observation**

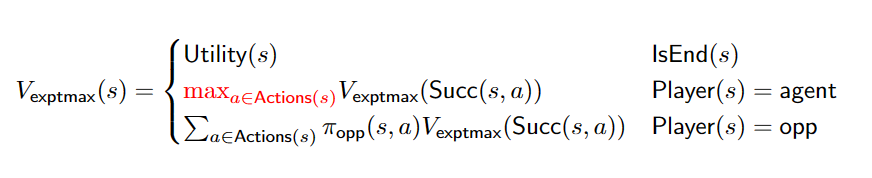
**-why pac man rushes to the closest ghost**

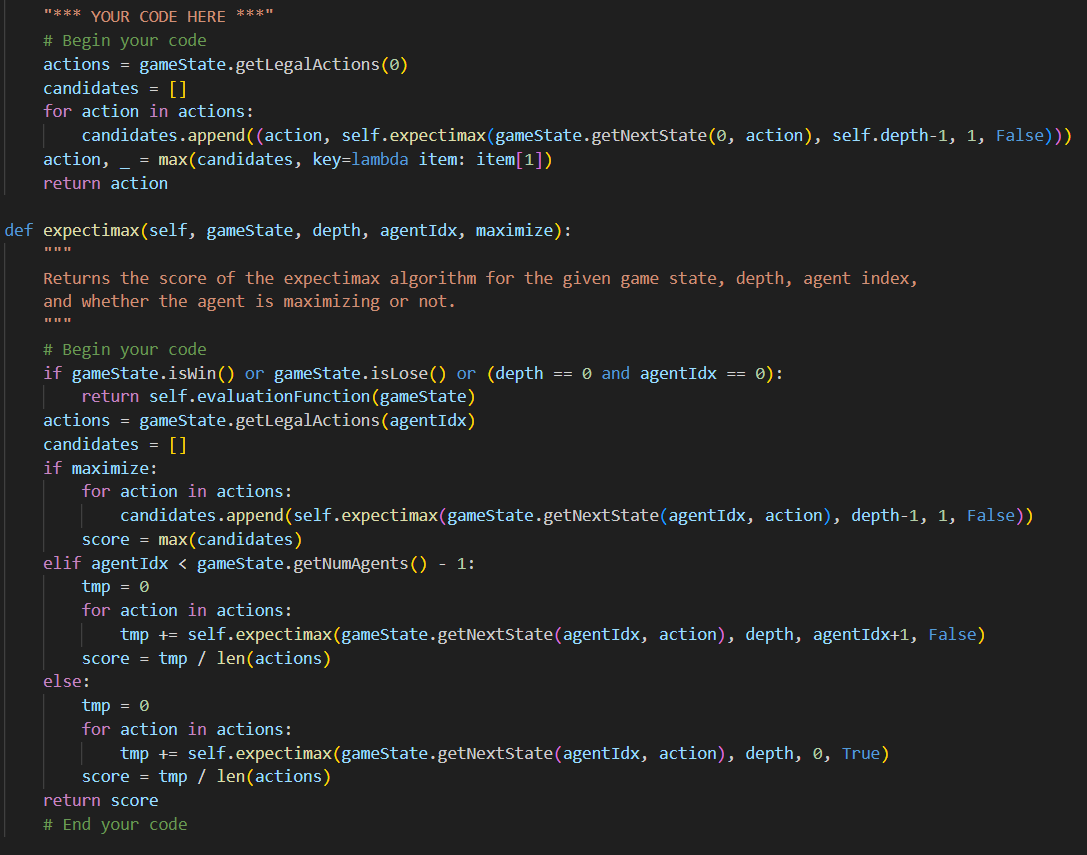
我想pacman會去撞右邊的橘鬼是因為minimax演算法會採最壞打算，也就是藍鬼會往右的方向走，因此若是pacman往左走選擇吃豆子則會和藍鬼撞到，因此選擇往右走。



**1.2 Expectimax**

Expectimax演算法是Minimax演算法的一種變形。在每個決策點上，考慮所有可能的行動，並計算每個行動的期望值。它假設對手的行動是以某種機率分佈選擇行動。因此，它不像Minimax演算法那樣假設對手會選擇最佳行動。





Expectimax演算法和Minimax演算法很像，只差在當為min layer時是取所有可能值的期望值，在此因為每一次行動的機率都是一樣的(隨機)，因此min layer的分數計算為所有動作的總分數的平均值。

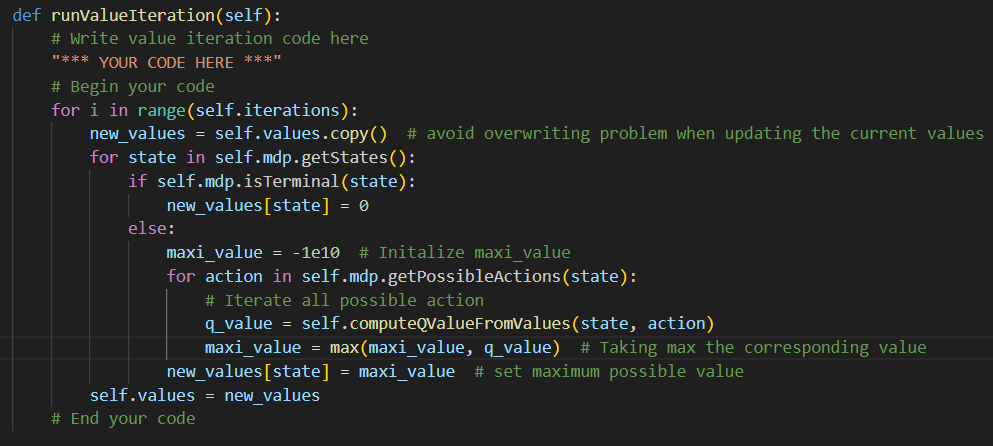
**Observation**

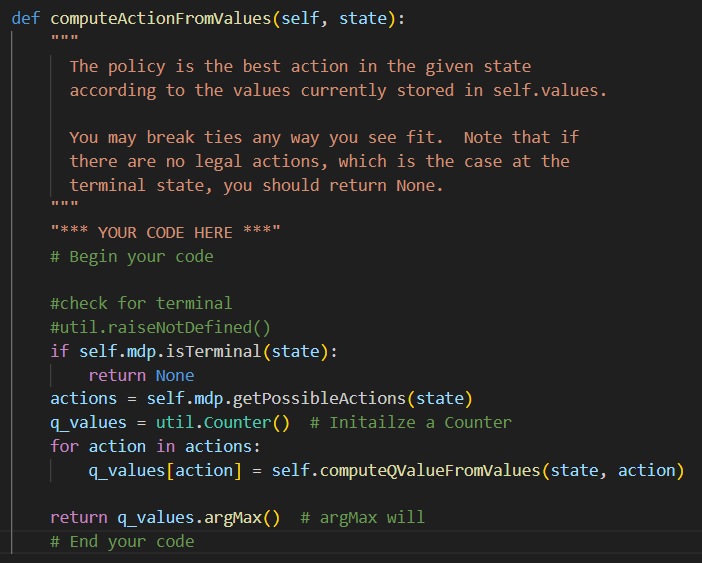
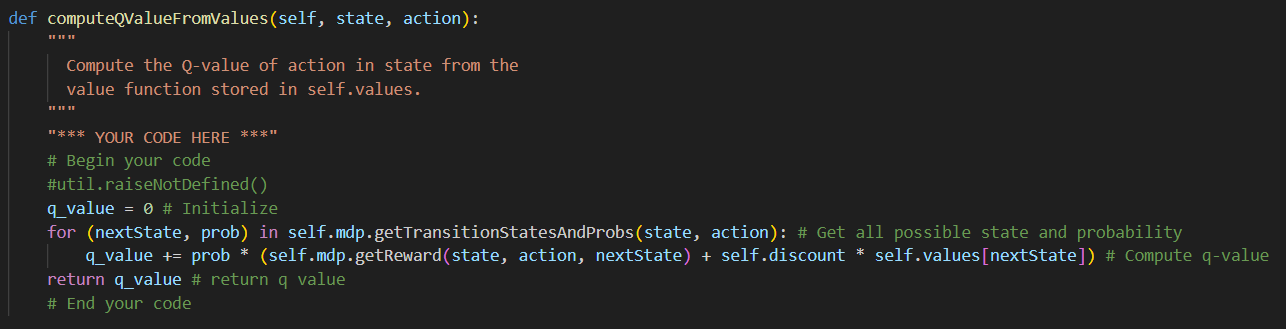
在Expectimax演算法中，pacman不會和Minimax演算法一樣開局就往右走。我想是因為pacman不是考慮最壞情況，而是考慮平均情況。所以如果藍鬼選擇往下。則將有機會吃掉那四個豆子。所以pacman會選擇往左走。

* **Part 2**

**2.1 ValueIterationAgent**

在這裡我們要在Markov decision process（MDP）的假設下計算每個狀態的價值函數來找出最佳策略。其表達式如下:





**runValueIteration**

new\_values用於存儲每個狀態的新值，避免覆寫。對於每個狀態，根據MDP的定義進行處理：如果狀態是終局狀態，將其new values設置為0。否則先將初始化maxi\_value為1e10(很小的值)。接著計算對應的q value並選擇最大值來更新maxi\_value。接著將maxi\_value設置為狀態的new values。更新self.values為new values，完成一次迭代。

**computeQValueFromValues**

根據給定的狀態和動作計算Q值。其計算方式如下。

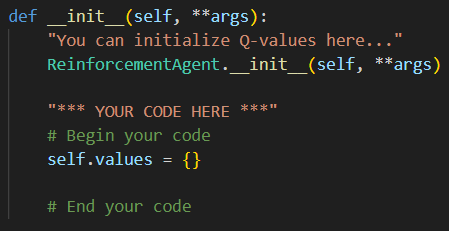
**computeActionFromValues**

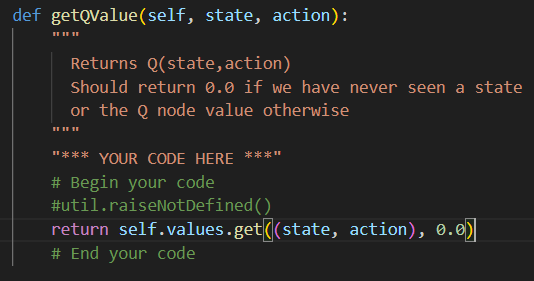
用來計算最佳動作。它對於每個可執行的動作計算對應的q value，並返回具有最大q value的動作。

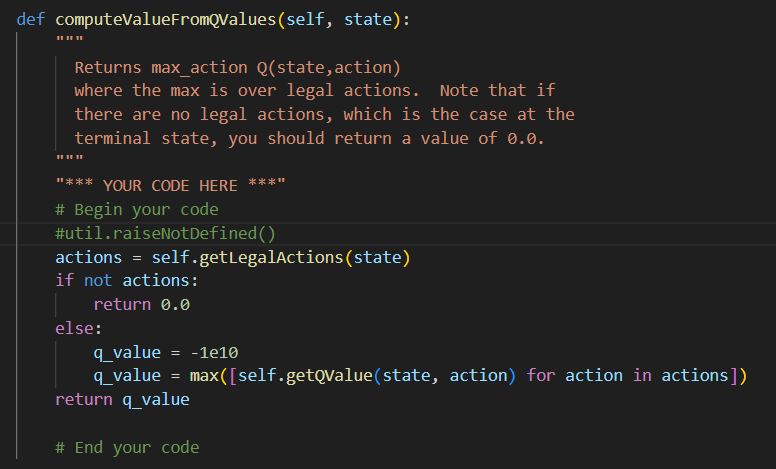
**2.2 & 2.3 Q-learning**

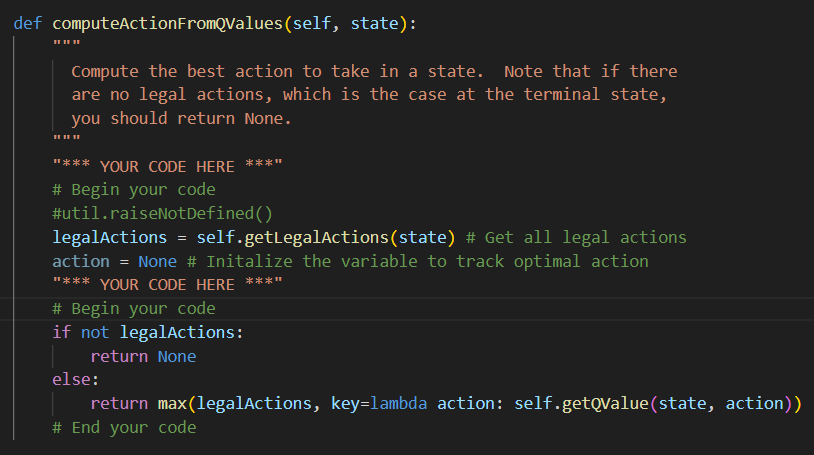
在這個部分要實作Q-learning with epsilon-greedy action。是在Q-learning演算法中引入一個機率參數epsilon，可以控制Agent選擇隨機行動的概率。目的通過在選擇行動時引入隨機性，避免Agent陷入局部最優解。在Q-learning with epsilon-greedy action中，Agent根據epsilon的概率選擇隨機行動，並以1-epsilon的概率選擇利用計算出的最佳行動。

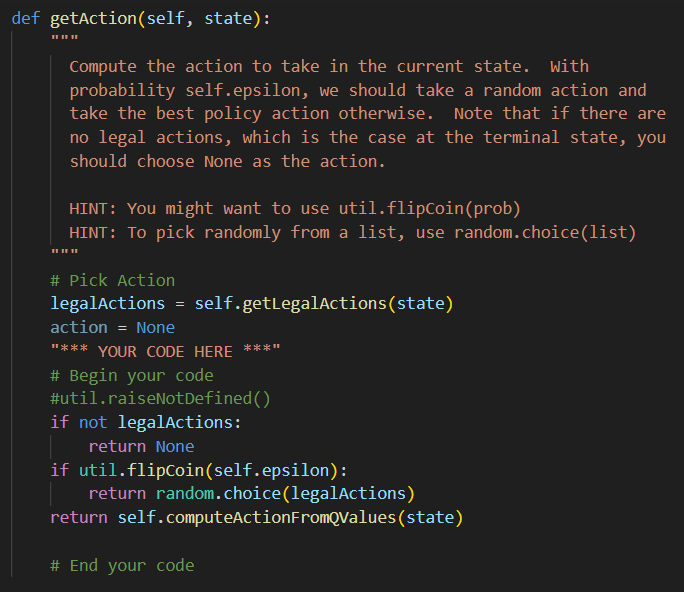
程式碼如下:

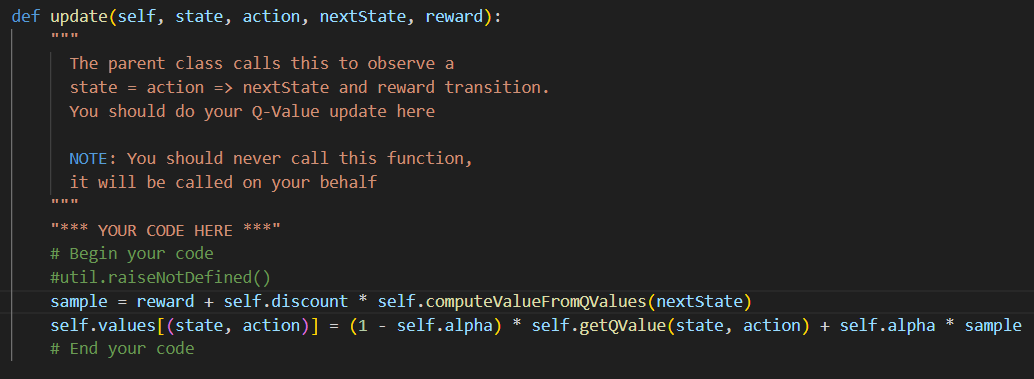












**\_\_init\_\_**

初始化了一個空的字典self.values，用於存狀態和行動的q value。 **getQValue**

根據給定的狀態和行動返回q value。如果該狀態和行動從未出現過，則返回0.0。

**computeValueFromQValues**

根據給定的狀態計算最大的q value，並返回。如果沒有合法的行動可選，則返回0.0。

**computeActionFromQValues**

計算在給定狀態下應該採取的最佳行動。如果沒有合法的行動可選，則返回None。

**getAction**

計算在當前狀態下應該採取的行動。根據機率(self.epsilon)，可能隨機選擇一個行動，或者根據q value選擇最佳行動。如果沒有合法的行動可選，則返回None。

**Update**

使用Q-learning的更新規則計算新的q value並存進self.values中。

**Observation**

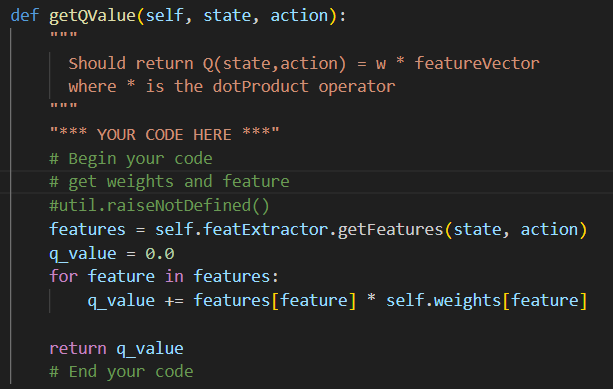
**-For different epsilon values. Does that behavior of the agent match what you expect?**

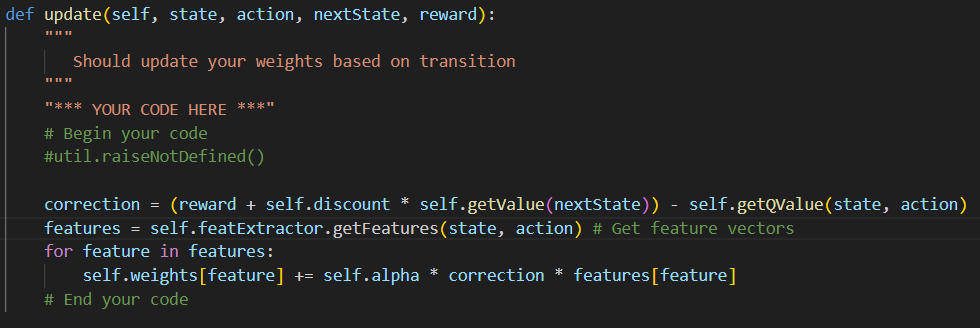
隨著epsilon的調高，藍點會動得更隨機，但跑到負分的機率也會增高。而調低後，藍點會動得很拘謹，但是隨著時間拉長，跑到正分的次數變多，之後的行動也變得很快，也都有一個固定的路線。而當調到0或非常接近0後是完全或幾乎不會動的狀態。

**2.4 Approximate Q-learning**

這個部分要實作Approximate Q-learning。在一般的Q learning中，我們使用一個Q table來存儲每個狀態和行動的q value。然而，當狀態空間非常大時，這種方法會不太可行，需要大量的記憶體和計算資源。 而Approximate Q learning通過使用一個特徵提取器（Feature Extractor）和權重（Weights）來近似表示q value。特徵提取器是一個用於將狀態和行動轉換為特徵向量的函數。這些特徵可以是關於狀態和行動的任何信息，如位置、距離等。特徵提取器的目的是捕捉對問題解決有用的特徵。這些特徵向量與一組權重進行內積，得到對應的q value。

在ApproximateQAgent中，只有2個函式需要實作：getQValue和update。程式碼如下:



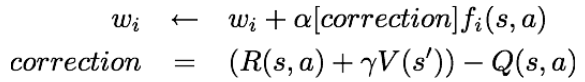


**getQValue**

計算給定狀態和行動的q value。首先使用feature extractor來獲取狀態和行動的特徵向量，然後將這些特徵向量與權重進行內積運算，得到q value的估計。

**Update**

用於根據觀察到的狀態轉換（state = action => nextState）和Reward來更新權重。其權重的表達式如下:



**Observation**

-**SimpleExtractor**

用於 Pacman 遊戲狀態的特徵提取。在獲取特徵時，提取器會從遊戲狀態中提取食物和牆壁的位置，並獲取幽靈的位置。然後根據 Pacman 執行的動作和其下一個位置的情況，計算以上特徵的值。最後再將特徵值進行歸一化。其中包含以下特徵：

* + **bias（偏差）**：這是一個常數特徵，用於提供一個偏差項，以幫助模型進行預測。
  + **#-of-ghosts-1-step-away（一步之遙的幽靈數量）**：計算 Pacman 在執行動作後，周圍一步距離內的幽靈數量。如果有幽靈靠近，該特徵的值將大於 0。
  + **eats-food（吃食物）**：如果在執行動作後，Pacman 的下一個位置有食物，則該特徵的值為 1.0，否則為 0。
  + **closest-food（最近的食物距離）**：計算 Pacman 的下一個位置到最近食物的歐式距離，並將其標準化為一個小於 1 的數字。

**Part 3**

1. **What is the difference between On-policy and Off-policy**

**Ans:** 這兩者本質區別在於更新q value時所使用的方法是沿用既定的策略（on-policy）還是使用新策略（off-policy）。On-policy方法使用相同的策略進行學習和探索。也就是說，在訓練過程中，它使用的行動策略（用於選擇行動）和目標策略（用於估計值函數）是相同的。例如: 使用ε-greedy策略同時進行探索和學習。Off-policy方法則使用不同的策略進行學習和探索。它使用的行動策略和目標策略是不同的。例如: 使用ε-greedy策略進行探索，但目標策略可能是根據已經學習到的最佳策略。

1. **Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function**

**Ans:**

* **value-based** 是指衡量在給定策略下，從特定狀態開始到結束(終止狀態)的回報。目標是找到最佳策略，並使價值函數(value function)最大化。
  + **value function** 大概長這樣:

其中 表示在策略π下從狀態𝑆開始的預期回報。 T (s, a, s') 是如果Agent處於遊戲狀態 s 並採取行動 a 時到遊戲狀態 s' 的轉移機率。Reward(s, a, s') 是給定當前遊戲狀態 s、動作 a 和下一個遊戲狀態 s' 的獎勵。γ 是discount factor，越大表示未來的影響越小

* **policy-based**是直接學習和優化策略本身，而不需要顯式地建模和優化價值函數。這些方法尋求找到一個最優策略π，使得在該策略下的長期回報最大化。
* **Actor-Critic**方法結合了value-based和policy-based。它同時學習策略和價值函數。Actor代表策略，根據當前策略選擇行動，Critic則是價值函數的估計器，用於評估行動的價值。Actor-Critic方法可以充分利用價值函數的估計，並根據該估計進行策略的改進。

1. **What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating .**

**Ans:**

* **Monte-Carlo**會通過模擬完整軌跡（從起始狀態到終止狀態）來估計價值函數的方法。它根據實際經驗軌跡的回報來更新狀態的值。每次更新時必須要等到一個episodes的結束。優點是可以在完整的軌跡上進行學習，但缺點是需要等到軌跡結束才能進行更新。
* **Temporal-difference**是即時獎勵和下一個狀態的估計值來更新狀態的值。TD可以從不完整的episode學習，效率比起MC更好但是也因為每次都在更新因此會有較高的variance，MC則幾乎不會有。

1. **Describe State-action value function and the relationship between in Q-learning.**

**Ans:**  表示從狀態 𝑆 開始並遵從策略 𝜋 時的預期回報。衡量在給定策略下處於特定狀態的價值。 表示在狀態 𝑠 下執行動作 𝑎 的預期回報。衡量在給定策略下在特定狀態下採取特定動作的價值。

兩者在Q-learning的關聯可以表示成:

1. **Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.**

**Ans:**

* **Target Network**：傳統的Q-learning使用單一的Q(s,a)來估計行動值。但容易導致目標值的不穩定性和震盪。Target network是一個與主網路（Q(s,a))結構相同的網路，用於計算目標值。在更新Q(s,a)時，使用target network的參數來計算目標值，從而增加算法的穩定性。
* **Exploration**：是指在學習過程中主動探索未知的狀態和行動，而不僅僅依賴於已知的知識。探索是為了發現新的信息，助於提高策略的品質。常見的探索策略包括 ε-greedy。
* **Replay Buffer**：用於存儲過去的經驗數據的緩衝區。在 Q-learning 中，我們將先前的狀態、行動、獎勵和下一個狀態存儲在replay buffer中。這樣做的目的是為了使學習更穩定，減少數據的相關性。在每次學習更新時，從replay buffer中隨機抽樣一批數據進行訓練，而不是即時使用當前的數據，從而減少數據之間的相關性，使學習更加平穩。

1. **Explain what is different between DQN and Q-learning.**

**Ans:** DQN（Deep Q-Network）是 Q-learning 的一個變體，其主要區別在於使用了DNN來近似 Q 函數的值。傳統的 Q-learning 使用表格來存儲和更新 Q function的值，而 DQN 則使用DNN來近似Q 函數。使得DQN 在處理具有高維狀態空間的問題時會更有效，並能夠推廣到更廣泛的情況。DQN 和 Q-learning 之間的不同之處大致上為以下：

**Experience Replay**：將過去的經驗存儲在buffer (replay buffer)中。上面有對於replay buffer的解釋。

**Target network**：上面提過了

* **Comparisons**

Different method comparison (100 games, 2 ghost, smallClassic):

(for q-learning method, ε = 0.05, γ = 0.8, α = 0.2)

|  |  |  |
| --- | --- | --- |
| **Method** | **Win rate** | **Average Score** |
| Minimax (depth = 3) | **0.64** | **776.5** |
| Expectimax (depth = 3) | **0.48** | **438.9** |
| Approximate Q-learning (trained 2000 episodes) | **0.87** | **834.7** |
| DQN (助教預訓練好的) | **0.89** | **1359.8** |

由上面的比較表可看出， 所有方法中DQN表現最好，approximate Q-learning次之，而Minimax的表現好於Expectimax。我認為是因為Minimax由於搜索深度有限，可能無法捕捉到太複雜的遊戲策略，因此勝率和平均得分相對較低。而Expectimax會把對手的行動視為隨機事件，並計算每個行動的期望值。也可能是因為搜索深度有限，導致勝率和平均得分相對較低。而Approximate Q-learning儘管只進行了2000個episodes，但它在勝率和平均得分方面表現得相對不錯。也可能是因為訓練回合數較少，無法完全學習到最佳策略，從而限制了其表現。DQN使用助教預訓練好的模型，應該意味著該模型已經在大量的訓練數據上進行了學習，並且具有更豐富的遊戲知識和策略。所以表現大於其他的方法。

以下是 DQN 相對於 Approximate Q-learning 的可能優勢和原因解釋：DQN 的勝率為 0.89，相較於 Approximate Q-learning 的 0.87。這表明 DQN 在遊戲中取得勝利的能力更強，可能更有效地學習到最佳策略。DQN 的平均得分為 1359.8，而 Approximate Q-learning 的平均得分為 834.7。這意味著 DQN 在遊戲中獲得更高的分數，顯示出更優秀的遊戲表現。我認為是DQN 使用DNN來近似 Q 函數，這使得它能夠處理更複雜的遊戲狀態和行動空間，可以學習到更細緻的遊戲策略和動作價值估計，這可能有助於 DQN 在遊戲中達到更高的勝率和得分。且DQN是用先預訓練好的模型，表示已經在大量訓練數據上進行了學習。這使得初始模型具有更好的遊戲知識和策略，有助於提高初始性能和快速學習最佳策略。

* **Problem I meet**

part 2-2 和 2-3實作的Q-learning似乎只能跑gridworld，找不到方式可以跑和其他方法一樣基準(100 games, 2 ghost, smallClassic)，他會顯示no attribute “getScore”，所以我就沒有比較Q-learning。

