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:

$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

```
    actions = gameState.getLegalActions(0)
    candidates = []
    for action in actions:
        candidates.append((action, self.minimax(gameState.getNextState(0, action), self.depth-1, 1, False)))
    action, _ = max(candidates, key=lambda item: item[1][1])
    return action

def minimax(self, gameState, depth, agentIdx, maximize):
    """
    Returns the (state, score) pair for the current gameState at a given depth and agentIdx.
    If maximize is True, then it is a max layer; otherwise, it is a min layer.
    """
    if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIdx == 0):
        return (gameState, self.evaluationFunction(gameState))

    actions = gameState.getLegalActions(agentIdx)
    candidates = []
    if maximize:
        # Max layer
        for action in actions:
            nextGameState = gameState.getNextState(agentIdx, action)
            candidates.append(self.minimax(nextGameState, depth-1, 1, False))
        stateScore = max(candidates, key=lambda item: item[1])[1]
    elif agentIdx < gameState.getNumAgents()-1:
        # Min layer with intermediate ghosts
        for action in actions:
            nextGameState = gameState.getNextState(agentIdx, action)
            candidates.append(self.minimax(nextGameState, depth, agentIdx+1, False))
        stateScore = min(candidates, key=lambda item: item[1])[1]
    else:
        # Min layer with the last ghost
        for action in actions:
            nextGameState = gameState.getNextState(agentIdx, action)
            candidates.append(self.minimax(nextGameState, depth, 0, True))
        stateScore = min(candidates, key=lambda item: item[1])[1]

    return (gameState, stateScore)
```

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。

$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

```
    "*** YOUR CODE HERE ***"
    # Begin your code
    actions = gameState.getLegalActions(0)
    candidates = []
    for action in actions:
        candidates.append((action, self.expectimax(gameState.getNextState(0, action), self.depth-1, 1, False)))
    action, _ = max(candidates, key=lambda item: item[1])
    return action

def expectimax(self, gameState, depth, agentIdx, maximize):
    """
    Returns the score of the expectimax algorithm for the given game state, depth, agent index,
    and whether the agent is maximizing or not.
    """
    # Begin your code
    if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIdx == 0):
        return self.evaluationFunction(gameState)
    actions = gameState.getLegalActions(agentIdx)
    candidates = []
    if maximize:
        for action in actions:
            candidates.append(self.expectimax(gameState.getNextState(agentIdx, action), depth-1, 1, False))
        score = max(candidates)
    elif agentIdx < gameState.getNumAgents() - 1:
        tmp = 0
        for action in actions:
            tmp += self.expectimax(gameState.getNextState(agentIdx, action), depth, agentIdx+1, False)
        score = tmp / len(actions)
    else:
        tmp = 0
        for action in actions:
            tmp += self.expectimax(gameState.getNextState(agentIdx, action), depth, 0, True)
        score = tmp / len(actions)
    return score
    # End your code
```

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:

$$V_{opt}^{(t)}(s) = \max_{a \in actions(s)} \sum_{s'} T(s,a,s') \left(Reward(s,a,s') + \gamma V_{opt}^{t-1}(s') \right)$$

```python
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    # Begin your code
    for i in range(self.iterations):
        new_values = self.values.copy()  # avoid overwriting problem when updating the current values
        for state in self.mdp.getStates():
            if self.mdp.isTerminal(state):
                new_values[state] = 0
            else:
                maxi_value = -1e10  # Initalize maxi_value
                for action in self.mdp.getPossibleActions(state):
                    # Iterate all possible action
                    q_value = self.computeQValueFromValues(state, action)
                    maxi_value = max(maxi_value, q_value)  # Taking max the corresponding value
                new_values[state] = maxi_value  # set maximum possible value
        self.values = new_values
    # End your code
```

```python
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    q_value = 0 # Initialize
    for (nextState, prob) in self.mdp.getTransitionStatesAndProbs(state, action): # Get all possible state and probability
        q_value += prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState]) # Compute q-value
    return q_value # return q value
    # End your code
```

```python
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code

    #check for terminal
    #util.raiseNotDefined()
    if self.mdp.isTerminal(state):
        return None
    actions = self.mdp.getPossibleActions(state)
    q_values = util.Counter()  # Initailze a Counter
    for action in actions:
        q_values[action] = self.computeQValueFromValues(state, action)

    return q_values.argMax()  # argMax will
    # End your code
```

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。

$$Q_{k+1(s)} = \sum_{s'} T(s,a,s')\left(Reward(s,a,s') + \gamma \max_{a'} Q_{k(s',a')}\right)$$

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:

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    # Begin your code
    self.values = {}

    # End your code
```

```python
def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    return self.values.get((state, action), 0.0)
    # End your code
```

```python
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    actions = self.getLegalActions(state)
    if not actions:
        return 0.0
    else:
        q_value = -1e10
        q_value = max([self.getQValue(state, action) for action in actions])
    return q_value

    # End your code
```

```python
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    legalActions = self.getLegalActions(state) # Get all legal actions
    action = None # Initalize the variable to track optimal action
    "*** YOUR CODE HERE ***"
    # Begin your code
    if not legalActions:
        return None
    else:
        return max(legalActions, key=lambda action: self.getQValue(state, action))
    # End your code
```

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    if not legalActions:
        return None
    if util.flipCoin(self.epsilon):
        return random.choice(legalActions)
    return self.computeActionFromQValues(state)

    # End your code
```

```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()
    sample = reward + self.discount * self.computeValueFromQValues(nextState)
    self.values[(state, action)] = (1 - self.alpha) * self.getQValue(state, action) + self.alpha * sample
    # End your code
```

__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:

```python
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get weights and feature
    #util.raiseNotDefined()
    features = self.featExtractor.getFeatures(state, action)
    q_value = 0.0
    for feature in features:
        q_value += features[feature] * self.weights[feature]

    return q_value
    # End your code
```

```python
def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #util.raiseNotDefined()

    correction = (reward + self.discount * self.getValue(nextState)) - self.getQValue(state, action)
    features = self.featExtractor.getFeatures(state, action) # Get feature vectors
    for feature in features:
        self.weights[feature] += self.alpha * correction * features[feature]
    # End your code
```

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:

$$w_i \leftarrow w_i + \alpha[correction] f_i(s,a)$$
$$correction = (R(s,a) + \gamma V(s')) - Q(s,a)$$

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。

2. **Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function $V^{\pi}(S)$**

   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 $V^{\pi}(S)$** 大概長這樣:

     $$V_{k+1}^{\pi}(\mathrm{s}) = \max_{a \in actions(s)} \sum_{s'} T(s,a,s')\big(Reward(s,a,s') + \gamma\, V_{\mathrm{k}}^{\pi}(s')\big)$$

     其中 $V_{\mathrm{k+1}}(\mathrm{S})$ 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。

3. What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating $\mathbf{V^{\pi}(S)}$.

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。

4. Describe State-action value function $\mathbf{Q^{\pi}(s,a)}$ and the relationship between $\mathbf{V^{\pi}(S)}$ in Q-learning.

Ans: $V^{\pi}(S)$ 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。 $Q^{\pi}(s,a)$ 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:

$$\mathrm{Q}_{opt}(s,a) = \sum_{s'} T(s,a,s') \left(Reward(s,a,s') + \gamma \, V_{\mathrm{opt}}(s') \right)$$

5. 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。

6. 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, $\varepsilon = 0.05$, $\gamma = 0.8$, $\alpha = 0.2$)

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。