HW3

109550167 陳唯文

Part1-1 Minimax search

```
def minimax(agent, depth, gameState):
    if gameState.isWin() or gameState.isLose() or depth ==self.depth:
        return self.evaluationFunction(gameState)
    if agent == 0: #maximize(pacman)
        value = float("-inf")
        actions = gameState.getLegalActions(0)
        for action in actions:
            value = max(value, minimax(1, depth, gameState.getNextState(0, action)))
        return value
    else :  #minimize(ghosts)
        value = float("inf")
        nextAgent = agent + 1
        if gameState.getNumAgents() == nextAgent:
            nextAgent = 0
        if nextAgent == 0:
            depth += 1
        actions = gameState.getLegalActions(agent)
        for action in actions:
            value = min(value, minimax(nextAgent, depth, gameState.getNextState(agent, action)))
        return value
```

這裡的變數 agent 是用來代表 pacman 和 ghost。

當 agent = 0 時，代表這是 pacman 的回合。所以在這個 if 裡面，要回傳的值是做出不同 action 後能得到的最大分數，對應到 minimax 中**最大化**自己的分數。

當 agent >= 1 時，此時是 ghost 的回合，由於 ghost 可能不只有一隻，所以這裡的 nextagent 是用來判斷是否已經讓全部的鬼進行它們的回合，如果沒有，就繼續進行下一隻鬼的回合；如果是，代表所有人都已經執行過，所以將 depth+1。而回傳的值則是能讓 pacman 得到的最小分數，對應到 minimax 中對手會**最小化**我們的分數。

而當遊戲已經贏了或輸了，又或是已經進行全部的回合，就回傳這時得到的分數。

```
best_value = float("-inf")
final_action = Directions.NORTH
actions = gameState.getLegalActions(0)  #pacman's action(first step)
for action in actions:
    value = minimax(1, 0, gameState.getNextState(0, action))
    if value > best_value:
        best_value = value
        final_action = action

return final_action
```

最後算出能得到的最大分數，並回傳得到最好的分數時執行的動作。

Part1-2 Expectimax search

```
def expectimax(agent, depth, gameState):
    if gameState.isWin() or gameState.isLose() or depth ==self.depth:
        return self.evaluationFunction(gameState)
    if agent == 0: #maximize(pacman)
        value = float("-inf")
        actions = gameState.getLegalActions(0)
        for action in actions:
            value = max(value, expectimax(1, depth, gameState.getNextState(0, action)))
        return value
    else :  #minimize(ghosts)
        nextAgent = agent + 1
        if gameState.getNumAgents() == nextAgent:
            nextAgent = 0
        if nextAgent == 0:
            depth += 1
        actions = gameState.getLegalActions(agent)
        value = 0
        for action in actions:
            value += expectimax(nextAgent, depth, gameState.getNextState(agent, action))
        value /= float(len(gameState.getLegalActions(agent)))
        return value
```

這邊和 minimax 很像，只是在輪到鬼的回合時，我們判斷鬼並不會執行對自己最有利的動作，而是隨機採取行動，所以這裡回傳的是鬼在執行不同動作後的分數期望值總和。而 pacman 的回合依舊是最大化自己的分數。

Part1-3 Evaluation Function

```
newPos = currentGameState.getPacmanPosition()
newFood = currentGameState.getFood().asList()
newGhostStates = currentGameState.getGhostStates()
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]
newCapsules = currentGameState.getCapsules()
scared = sum(newScaredTimes)

food_dist = 50

for food in newFood:
    food_dist = min(food_dist, manhattanDistance(newPos, food))

food_dist = 1.0/food_dist


cap_dist = 50

for capsule in newCapsules:
    cap_dist = min(cap_dist, manhattanDistance(newPos, capsule))

cap_dist = 1.0/cap_dist


return currentGameState.getScore() + food_dist + cap_dist + scared
```

這裡的策略是：離最近的食物越近，加越多分。離最近的膠囊越近，加越多分。最後再加上 scaredTimer。這樣的組合會造成 pacman 在膠囊前等鬼靠近，並馬上吃掉的情況，而且實測之後能夠達到 100%的勝率。

Part2-1 Value iteration

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    # Begin your code
    for i in range(self.iterations):
        currentValues = self.values.copy()
        states = self.mdp.getStates()
        for state in states:
            if not self.mdp.isTerminal(state):
                action = self.computeActionFromValues(state)
                currentValues[state] = self.computeQValueFromValues(state, action)
        self.values = currentValues
    # End your code
```

Input: MDP $(\mathbb{S}, \mathbb{A}, \mathrm{P}, R, \gamma, H)$
Output: $\pi^*(s)$'s for all s's

For each state s, initialize $V^*(s) \leftarrow 0$;
for $h \leftarrow 0$ **to** $H-1$ **do**
  foreach s **do**
    $V^*(s) \leftarrow \max_a \sum_{s'} \mathrm{P}(s'|s;a)[R(s,a,s') + \gamma V^*(s')]$;
  end
end

這一部分對應到的是 value iteration 的這邊，利用動態規劃的概念，也就是每一個 state 的在當前的最佳報酬會利用上一步的下一個 state 得到的最佳報酬。

```
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
    value=0
    for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action):
        value += prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
    return  value
    # End your code
```

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s'|s) \left(R_a(s,s') + \gamma V_i(s') \right) \right\}$$

而這一行公式則是寫在 computeQvalueFromvalue 內，這樣寫可以確保在算第 k

次 iteration 時，使用的是第 k-1 次 iteration 的值

```
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
    if self.mdp.isTerminal(state):
        return None

    max, bestAction = float("-inf"), None
    for action in self.mdp.getPossibleActions(state):
        temp = self.computeQValueFromValues(state, action)
        if temp > max:
            max = temp
            bestAction = action

    return bestAction
```

最後的這個部分跟前面很像，就是根據當前的 V(s)值得出要採取的最佳策略

Part2-2 Q-learning

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

    "*** YOUR CODE HERE ***"
    # Begin your code
    self.qvalues = util.Counter()
    # End your code
```

使用 util.counter，這樣即使出現沒用過的 key，依然會回傳 0

```
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
    return self.qvalues[(state, action)]
    # End your code
```

之後要取得 qvalues 時以呼叫 function 取代直接呼叫 qvalues

```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
    if self.getLegalActions(state) is None:
      return None

    max = float("-inf")
    for action in self.getLegalActions(state):
      if self.getQValue(state, action) > max:
        max = self.getQValue(state, action)

    temp =[]
    for action in self.getLegalActions(state):
      if self.getQValue(state, action) == max:
        temp.append(action)

    result = random.choice(temp)

    return result
```

回傳能得到最大 qvalue 的動作，加上 random.choice 的目的是為了當其他 action 得到的值都是負的時候，將還沒採取過的動作(此時的 qvalue 為 0)平均分配機率，以得到更好的表現。

```python
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
    learned_value = reward + self.discount * self.computeValueFromQValues(nextState)
    self.qvalues[(state, action)] = (1 - self.alpha) * self.qvalues[(state, action)] + self.alpha * learned_value
    # End your code
```

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

這一段的 code 就是對應到 Q-learning 演算法的公式，self.alpha 就是學習速率 α，self.discount 是折扣因子γ。當γ越大，模型便會越注重以前得到的經驗，也就是長期獎勵，反之，模型會更看重眼前得到的報酬。和 value iteration 不一樣的地方是，Q-learning 會因應環境而做出改變。

Part2-3 epsilon-greedy action selection

```
# Begin your code
if len(legalActions) == 0:
  return None
if util.flipCoin(self.epsilon):
  action = random.choice(legalActions)
else:
  action = self.computeActionFromQValues(state)

return action
# End your code
```

這裡如同 spec 描述的一樣，利用 flipcoin 決定要回傳的動作

Part2-4 Approximate Q-learning

```
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
    w = self.getWeights()
    featureVector = self.featExtractor.getFeatures(state, action)

    return w * featureVector
```

$$Q(s,a) = \sum_{i}^{n} f_i(s,a) w_i$$

按照指示和公式，回傳 qvalue = w * featureVector

```
def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    correction = reward + self.discount*self.computeValueFromQValues(nextState) - self.getQValue(state, action)
    featureVector = self.featExtractor.getFeatures(state, action)
    for feature in featureVector:
      self.weights[feature] += self.alpha*correction*featureVector[feature]

    # End your code
```

$$
\begin{aligned}
w_i &\leftarrow w_i + \alpha[correction] f_i(s,a) \\
correction &= (R(s,a) + \gamma V(s')) - Q(s,a)
\end{aligned}
$$

依照 spec 提供的公式更新權重。

在執行 autograder.py part2-4 的部分時，我並沒有拿到全部的十分，然後在觀察小精靈實際執行的 graphic 後，我發現小精靈幾乎不會去吃膠囊，而在 featureExtractor 中也的確沒有關於膠囊的特徵值，所以如果要取得更高的分數以及更好的勝率，應該可以在 featureExtractor 中加上關於膠囊的資訊。

Part3 DQN

```
Average Score: 1262.2
Scores:        315.0, 1743.0, 1031.0, 1537.0, 957.0, 1556.0, 1771.0, 1354.0, 1168.0, 1166.0, 84.0, 1558.0, 1568.0, 1120.
0, 1745.0, 1776.0, 1548.0, 1763.0, 1552.0, 1347.0, 1688.0, 1128.0, 245.0, 1769.0, 1569.0, 1534.0, 696.0, 1526.0, 1577.0,
 1722.0, 1750.0, 1354.0, 1151.0, 1427.0, 1747.0, 1302.0, 594.0, 1538.0, 1365.0, 1728.0, 1116.0, 1371.0, 303.0, 936.0, 66
2.0, 1535.0, 30.0, 1339.0, 728.0, 1679.0, 1337.0, 1273.0, 1521.0, 1358.0, 1148.0, 957.0, 1034.0, 1038.0, 1571.0, 1311.0,
 1509.0, 1174.0, 1715.0, 1452.0, 665.0, 1540.0, -112.0, 1463.0, 1730.0, 1555.0, 1559.0, 335.0, 1345.0, 1125.0, 1740.0, 1
370.0, 1110.0, 1712.0, -300.0, 1723.0, 1120.0, 1366.0, 1332.0, 1407.0, 1775.0, 1174.0, 1566.0, 1296.0, 87.0, 1108.0, 165
6.0, 1538.0, 1528.0, 1371.0, -129.0, 1506.0, 1366.0, 1514.0, 1382.0, 1531.0
Win Rate:      86/100 (0.86)
Record:        Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Win,
 Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Loss,
Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Wi
n, Win, Win, Win, Loss, Win, Win, Win, Win, Win
```

執行 python pacman.py -p PacmanDQN -n 200 -x 100 -l smallClassic 後的結果

Comparison

地圖 ：smallClassic

Method	Average score	Win rate
Expectimax (depth=3)	557.61	55/100(0.55)
Q-learning	823.48	86/100(0.86)
DQN	1262.2	86/100(0.86)

從實測之後的結果可以發現，雖然 expectimax 在 part1-3，只有一隻鬼情況下有 100%的勝率，而且平均分數達到 1200 分，但到了同時有兩隻鬼的地圖中，平均分數跟勝率都下降了一大截，可能是因為對於兩隻鬼，系統需要考慮更多的

資訊以及更複雜的情形，所以即使在 depth=3 之後，依舊沒辦法有效處理以得到更好的表現。
Q-learning 跟 DQN 的勝率則是不相上下，可能是因為在 part2 的 autograder.py 指令中讓 Q-learning 訓練了 2000 次，但 DQN 只訓練了 100 次。不過即使這樣，DQN 的平均分數依舊是遠超過單純的 Q-learning，這可能是因為 DQN 是使用深度學習結合 Q-learning，利用神經網路取代 Q-learning 的 Q 值表，所以能夠處理比之前更大量的資訊和結果，也因此能夠得到更好的表現。

Problems

在寫這次作業時我覺得最麻煩的就是要先搜尋很多資料並讀懂，包括 minimax 的運作原理，還有 value iteration 跟 Q-learning 的公式和公式代表的意義、思考邏輯，才能知道這次的每個 part 要寫什麼。尤其在 part2 的時候，光是要讀 spec 裡面的內容就要花很長時間了，還需要再從網路上找尋相關資料，才了解我們要實作的是那些部分。而我覺得最可惜的是在 part2-4 沒有拿到滿分，感覺如果有多新增關於膠囊的特徵值，應該可以取得更高的分數。但這次的作業確實讓我更加了解 Q-learning 和 minimax 的原理跟實際應用的方面。